Building High-Performance Web-Caching Servers

PhD Thesis

Alex Arshinov

De Montfort University

2004

ABSTRACT

World-Wide Web is one of the primary applications of Internet today. Web-caches can decrease bandwidth consumed by HTTP traffic and improve user experience by decreasing Web object retrieval latencies. Transparent web-caches can be used by organisations to intercept and cache all HTTP traffic without significant administrative expenses and therefore minimize traffic costs and improve filtering and monitoring capabilities. Internet Service Providers use en-route transparent Web-caching on their backbone communication links to decrease amount of HTTP traffic, which currently represents a major part of overall traffic. Web-caches are used in Content Delivery Networks to push content closer to end user, greatly improving latencies of object retrieval and reducing overall Internet traffic, at the same time offloading original Web-server. Web-caches are used in accelerator mode in high-volume Web-sites, decreasing overall cost of the Web-site and/or improving its scalability and performance. Though web-caches are often invisible at first glance, modern Internet in the form we see it today would be probably not possible without wide deployment of Web-caching technology. Deployment of web-caching technology gives immediate effect; performance of properly designed web-caching system can be improved step-by-step as needed.

Building high-performance web-caches capable of serving multigigabit links is a challenging task. Web-caching system must be able to handle up to 16 000 user requests per second per every gigabit of traffic (for average object size of 8KB, 1Gbps = $2^{30}/(8 \cdot 8 \cdot 2^{10}) = 2^{14}$ requests per second). Total cache size must allow to store up to several days of HTTP traffic, which is impossible to store in RAM (due to the overall price considerations), and therefore the performance and capacity of persistent storage (typically hard drives) becomes a crucial issue, because hard drives are mechanical devices and are not capable of sustained rate of random read-write operations exceeding a few hundred operations per second. Often web-caching system is the only possibility for end user to fetch web-object (even though user may not be aware of it, because all HTTP traffic will be diverted at router to transparent web-cache), and therefore reliability and failover capabilities of web-caching systems must meet strictest requirements. Last but not least web-caches must be easy to use and administer, ideally

not requiring any actions on behalf of end user, and require a minimum effort from network administrators.

One of the most promising solutions to the problem of web-cache scalability is web-cache clustering. Clustering is a technology of building from a few building blocks (e.g. low-cost PCs) a single virtual object visible from outside as single entity. Clustering is widely used in high-performance computations, high-performance web-servers and databases, and everywhere where a single big computational or processing task can be parallelized and run simultaneously at several computers. Web caching is no exception – single image web-caching clusters have the best price/performance ratio possible and provide almost linear scalability, easy extendibility, good reliability, seamless failover capabilities, and require little maintenance.

In this thesis a particular implementation of web-caching cluster is proposed which is capable of handling 500Mbps of HTTP traffic at the cost less than $20000 using cheap PC hardware, high-quality open-source software. As shown below, proposed solution has a number of advantages to other methods of solving web-cache scalability problems. First, the proposed cluster scheme is designed from very beginning to take into account the properties and characteristics of web-caching application as distinct from universal approach to clustering. One of the most important differences between cached content and, say, database records is the fact that cached data can be discarded at any moment, and this will not have any grave consequences in contrast to lost database records. Second, proposed approach to cluster building does not require any additional hardware (well, probably additional network switch), changes to application web-caching software or changes to operating system. Proposed cluster architecture is entirely software-based. Third, this particular implementation of cluster uses transparent web-caches (therefore any configuration at user's side is not needed), and cluster control software is easy to install and operate, require only a minimal effort from network administrator, at the same time providing good performance, excellent scalability and automatic fault detection and failover capabilities.

ACKNOWLEDGEMENTS

I would like to express my deepest thanks to my supervisors Prof. Hongji Yang and Prof. Hussein Zedan for their advice, encouragement and incredible patience during my last year of study.

I would like to thank my initial supervisors Prof. Jonathan Blackledge, Prof. Brian Foxon, Prof. Yury Kirchin and Dr. Andrey Robachevsky for giving me right direction of study and encouragement during my first years.

I would also like to thank my wife Tanya for her love and patience.

DECLARATION

I hereby declare that the work described is entirely mine and original and was undertaken from August 1998 to September 2004. It is submitted for the degree of Doctor of Philosophy at De Montfort University.

CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

AMD	Advanced Micro Devices
ATA	AT-attached
ATM	Asynchronous Transfer Mode
BSD	Berkeley Software Distribution
CPU	Central Processing Unit
CDN	Content Delivery Network
CGI	Common Gateway Interface
CSMA/CD	Carrier Sense Multiple Access/Collision Detection
CSP	Cluster State Protocol
CSS	Cascading Stylesheets
DNS	Domain Name System
FFS	Fast Filesystem
FTP	File Transfer Protocol
GB	Gigabyte (1024 Megabytes)
Gbps	Gigabits per second
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICP	Internet Cache Protocol
IDE	Integrated Device Electronics
IP	Internet Protocol
KB	Kilobyte (1024 Bytes)
LAN	Local Area Network
LRU	Least Recently Used
LBD	Load Based Distribution
MAC	Media Access Control
MAN	Metropolitan Area Network
MB	Megabyte (1024 Kilobytes)
Mbps	Megabits per second
NLANR	National Laboratory for Applied Network Research
NTFS	New Technology Filesystem

OS	Operating System
PAE	Physical Address Extension
PC	Personal Computer
PF	Packet Filter
PIF	Propagation of Information with Feedback
PRNG	Pseudo Random Number Generator
RAID	Redundant Array of (In)expensive Disks
RAM	Random Access Memory
RLD	Random Load Distribution
RPM	Rotations per minute
RRD	Round-Robin Load Distribution
RRP	Redundant Ring Protocol
RUNNet	Russian University Network
SCSI	Small Computers System Interface
SSL	Secure Sockets Layer
TB	Terabyte (1024 Gigabytes)
TCO	Total Cost of Ownership
TCP	Transmission Control Protocol
TTL	Time to Live
UDP	User Datagram Protocol
UFS	Unix Filesystem
URL	Universe Resource Locator
VFS	Virtual Filesystem
WAFL	Write Anywhere File Layout
WAN	Wide-Area Network
WCCP	Web Cache Coordination Protocol

CHAPTER 1

INTRODUCTION

1.1 Research Objectives and Overview of Problem

The global network – the Internet – has experienced a tremendous growth in years since introduction of the first real Web browser – NCSA Mosaic – in 1993. The World Wide Web, which once was just a toy, quickly became an essential part of life for most people in developed countries and is used nowadays for various applications from news distribution and communication between people to Internet banking, online shops and distant learning. A big and growing portion of overall Internet traffic is consumed by the HTTP protocol which drives the World Wide Web. And this tendency is likely to stay in the future, because there are still quite significant numbers of people in the world who do not have Internet connectivity at all or use Internet very rarely, so there is a big potential of traffic growth.

Internet communication links are an expensive resource which should be used wisely. One of the most natural ways of improving efficiency of particular communication links and the Internet as a whole is optimization of bandwidth consumption by HTTP protocol. One of the most important properties of WWW traffic is that a lot of people access the same information (popular news and sport sites etc.), and therefore it is not wise nor efficient to transfer the same web objects via the same communication link. Web-caches serve as the intermediaries between Web users and Web servers. By intercepting all Web requests Web-caches (sometimes called Web-proxies) optimize duplicate requests by storing first request in its storage (usually called cache) and serving consecutive requests not from original Web-server, but from Web-cache's storage. Therefore Web-caches decrease number of requests to remote Web-server, saving bandwidth, decreasing latency of

web-object retrievals, decreasing load to remote Web-server, thus benefiting not only users of the Web-cache, but the Internet as a whole. To be useful, Web-caches must meet a few requirements: they must be as simple to use as possible, and they must be reliable and provide at least the same or better performance (web-object retrieval times) as direct connection to the original Web-server. As it discussed below, all these requirements just sound easy, but meeting them is a demanding and complex task. The main focus of this thesis is performance of web-caches, and reliability is a secondary objective (though still very important one). In real life the price/performance ratio usually plays a crucial role in selecting the right method to solve the problem, so to be practical, high-performance web-cache must provide not only high-performance, but also at least competitive price/performance ratio. In this thesis these practical considerations are not left behind either. Every research is more convincing when it is backed with experiment or practical experience. Thus the objectives of this thesis were not only to propose a pure theoretical solution to web-caches scalability, but to provide a ready practical solution (special software and a method of building high-performance reliable web-cache from ready building blocks).

Modern high-speed communication links require powerful web-caches to handle a big and growing amount of HTTP traffic. The simplified algorithm of Web-cache is easy to understand: get request from user, check whether the requested object is in cache and is up-to-date, if yes then return object to the user, if not then fetch object from remote web-server, store object in cache and then return object to the user. Potential bottlenecks of every Web-cache are its capability to handle a large amount of TCP connections and HTTP requests, and the capability to store an optimal quantity of web-objects in cache *and* quickly locate a given object in cache. The number of simultaneous connections to web-cache is limited by Web-cache CPU, amount of available RAM and the sophistication of Web-cache's operating system and in particular its TCP/IP stack and filesystem used to store cached objects. Recent advances in CPU technology have allowed very-high-performance CPUs to

be used in cheap commodity PCs, and therefore computational resources rarely limit Web-caches (provided reasonable efficiency of web-caching software). RAM is quite cheap these days as well, but still RAM prices are prohibitive to build a completely RAM-based web-cache. TCP/IP stacks in modern Unix (FreeBSD, Solaris) and Unix-like (Linux) operating systems have proven itself to be highly robust and efficient. A single commodity PC running e.g. FreeBSD operating system could handle several tens of thousands of simultaneous TCP connections, so limits of TCP/IP stack will not become Web-cache performance bottleneck (at least not in first or second place). RAM is still not adequate to Web-cache requirements, because it is either not possible[1] or not practical[2] to store all cache's contents in main memory, so secondary storage such as hard disks must be used. Still due to the need to quickly search a required object in cache every Web-cache must have enough RAM to store at least some essential information about objects in cache, with every object requiring a couple hundred bytes in RAM, and therefore limiting maximum number of objects every given cache can store. Storage requirements leave no choice but to use some sort of persistent storage such as flash memory of hard drives. Flash drives (electronic power-independent persistent storage), though being a very promising technology, are still at the early stages of its development and their price is about the same as the price of RAM, leaving no choice but to use good old hard drives. Hard drives are mechanical devices, and it is the main reason

---

[1] Due to the lack of required number memory slots, prohibitive pricing of high-density memory modules, limits of computer's main boards, or 4GB limit of Intel's 32-bit PC architecture – PAE and other abuses do not count.

[2] Yes, it is possible to buy a Sun SPARC or Intel Itanium or AMD Opteron-based server and use 2GB or 4GB memory modules, but price/performance ratio will be horrible in comparison with commodity PCs with either Intel Pentium IV or AMD Athlon CPUs.

of their disadvantages. Hard drives provide excellent capacity (up to 400GB per unit at present time) and good performance for sequential read and write operations (up to 50MBps at present time), but for random access operations their performance is abysmal, rarely exceeding 300-400 random access read/write operations per second. Needless to say, web-caches use mostly random access operations when dealing with persistent storage. Therefore the maximum possible number of objects which can be served by given web-cache per second is the combined number of random access read/write operations per second achieved by hard drives of given web-cache. In reality, this number is lower, because storing or reading of object from cache may require more than one operation[3]. The number of hard drives in a single caching unit is limited by capabilities of disk interface. For commodity PCs this number is usually 4 (for Parallel and Serial ATA), servers may have SCSI interface with a maximum number of disks 14 per channel (though price/performance of SCSI disks will be worse in comparison with ATA drives). Still, the number and capacity of hard drives must correspond to the amount of RAM of Web-caching unit.

Performance of web-caches can be improved in several ways. First, it is possible to add additional RAM and hard drives, but beyond some point this approach becomes quite ineffective. Second, it is possible to use higher-density memory modules and/or faster drives (e.g. 15000RPM SCSI drives instead of 7200RPM IDE drives), but it comes at a price. Third, it is possible to optimize operating system and web-caching software to more efficiently use available resources. Possible improvements include better multithreaded TCP/IP stack with zero copy between user and kernel

[3] For general-purpose filesystems (such as FFS in FreeBSD or ext3fs in Linux), not optimized for web-caching scenario, the number of read/write operations to get/store web-object may be up to several tens [Danzig][Lee][Ganger].

mode, partial offloading of some computationally-intensive tasks to hardware (TCP and IP packet validation, checksums computation etc.), more efficient use of available RAM, and last but not least better filesystems, specifically designed for web-caching applications, which minimize number of disk read/write operations probably at the cost of absence of some features such as access rights and some other file attributes, useless for the purpose of web-caching. Still, the performance of a single web-caching unit has limits, and these limits are lower than requirements to web-caches in many situations.

Another approach to solving web-cache performance problems is to deploy a set of web-caching units instead of a single one. This causes a few problems, namely web users must be (often manually) assigned to each web-caching unit, which increases maintenance expenses; the contents of caches of web-caching units are duplicated, and not used wisely; there are no load sharing mechanisms in such groups of caches; overall reliability is rather low, because there are no failover mechanisms and computers sometimes break down or fail in some other way. Cache intercommunication protocols, such as ICP or HTCP, partially solve the problem of web-cache duplication and improve overall performance of a set of web-caches, but other problems still persist. ICP also limits scalability, because each web-cache in a group must communicate with each other, and handling ICP requests takes significant share of resources, making deployment of more than 10 cooperating web-caches impractical[4]. ICP can also be used to combine web-caches in tree-like hierarchies, and the higher given web-cache is located in hierarchy, the higher

[4] ICP traffic grows exponentially with number of Web-caches, at the same time increasing load to each participating web-cache. This limitation of ICP has been observed in real-life environment in NLANR caching hierarchy [NLANR].

performance and reliability requirements are applied to it. Cisco's WCCP protocol, designed to be used in interaction between routers (or Layer-3 switches) and web-caches, partially solves load balancing and web object space partitioning task, but WCCP is still far from perfect and requires an expensive Layer-3 switch (or more powerful router capable of running WCCP).

Traditional brute-force ways of improving processing capacity of web-caching units by using more powerful CPU, bigger number of CPUs, more operating and persistent storage or using more caches in web-caching hierarchy are not cost-effective and do not meet reliability requirements. Using groups of co-operating web-caches partially solves scalability problems, but also have significant disadvantages due to the absence of single cluster image and due to the need of co-operating web-caches to communicate with each other. Clustering is an effective way of solving these shortcomings of traditional solution to performance problems, but universal clustering methods do not take into account web-caching unique properties and characteristics. One of the most important differences between web-cache data and most of other types of data e.g. databases is that web-cache data could be discarded at any time without grave consequences (though performance of web-cache may suffer). Therefore duplication of web-cache data is usually not needed and even hurts a performance of web-cache cluster. A specialized solution to building high-performance web-cache clusters is needed, which would be more efficient, reliable, and easier to manage and operate and will have best possible price-performance ratio. In this thesis a new original web-cache clustering method is proposed which fully meets high performance, scalability and reliability requirements.

1.2 Scope of the Thesis and Contribution to Knowledge

This thesis deals exclusively with web-caching technology and its applications. Most important fields of web-caching technology are covered; most promising

directions are discussed more thoroughly. The focus of this thesis is performance and scalability problems of web-caches. Other important fields such as HTTP protocol modifications, better web-caching software (including multithreaded design and more efficient replacement algorithms), web object prefetching, optimized filesystems and other OS improvements are only touched briefly. One of the main objectives of this research was to be as practical as possible.

The contribution of this thesis is a new original method for building high-performance web-caches of virtually any processing capability using clusters of web-caches plus original cluster control software. It provides following benefits:

1) A single cluster image – cluster of many nodes is virtually undistinguishable from the outside from one powerful computer. This significantly simplifies configuration of nearby router, because its configuration is done only once, and new nodes could be added, removed or suspended transparently, no reconfigurations at the router are needed.
2) No additional hardware is required – everything is done via special configuration at the router and by using original cluster software. To convert existing group of co-operating web-caches to web-caching cluster a trivial operation is required (basically assigning new IP and MAC addresses at the nodes plus installation of cluster software).
3) Existing web-caching units can be used without any changes; Operating System and Web-caching software do not require any modifications (cluster software itself only consists of a small kernel loadable module plus a couple of usermode control processes).
4) Web-caching software at different nodes does not need to co-operate with each other, thus this overhead is completely eliminated. Cluster control software at different nodes communicates with other nodes using broadcast mechanisms; this overhead is minimal and virtually unnoticeable at any reasonably new computer.

5) Automatic load-balancing and failover capabilities are provided, which do not require any human intervention. Overall, cluster requires very little maintenance; most failures such as web-caching software faults, some types of hardware failures, loss of connectivity with some cluster nodes will be fixed automatically, and failed nodes will be put offline to be fixed later by system administrator; cluster's users will not be affected.
6) Cluster has excellent scalability, with the addition of new nodes performance grows almost linearly. The practical performance limit of the proposed cluster at present time is around 500Mbps, if higher performance is needed, several clusters could be combined in single one using trivial mechanisms such as round-robin distribution at the router[5].

1.3 Criteria for Success

The following criteria will be used to judge the success of research presented in this thesis:

1) Which benefits does the proposed clustering scheme provide in comparison to other methods of building high-performance web-caches? Why commercial proprietary solutions to web-cache performance are not sufficient?
2) Does the proposed web-cache cluster meet its design goals? Does it satisfy performance and reliability requirements? Does it provide required load-balancing and failover capabilities? Why these design goals are important?

---

[5] Cisco routers, for example, can divert all HTTP traffic to several transparent web-caches. This can be implemented in several ways, probably the simplest one would be manual balancing – one subset of IP addresses is diverted to first transparent proxy, second set – to another one and so on.

3) Are there any networks where proposed web-cache cluster would be really useful? What is the expected location in the network of web-caching cluster?
4) Is the proposed approach practically feasible? Can it be implemented in real network in such a way that any significant effort from system administrators and end users would not be required?

1.4 Thesis Structure

Thesis is organized in the following way:

1) Chapter 1 contains introductory material, including overview of field of research, gives basic description of research objectives, describes contribution of this thesis to knowledge, and provides information about thesis' structure.
2) Chapter 2 gives detailed overview of web-caching, including but not limited to benefits of web-caching, history of web-caching and miscellaneous applications of web-caching including transparent caches and Content Delivery Networks.
3) Chapter 3 describes recent trends in web-caching technology research which is mostly directed on improving end user experience by decreasing web-object retrieval latency, improving hit-rates by caching dynamic content and optimizing cache algorithms.
4) Chapter 4 gives overview of challenges of building high-performance web-caches, states requirements to high-performance web-caches and analyzes application of such web-caches to a particular network – Russian University Network.
5) Chapter 5 describes general cluster approach to web-caching, gives overview of various cluster protocols and technologies, analyzes advantages and disadvantages of particular approaches.
6) Chapter 6 provides thorough description of proposed approach of building high-performance web-caches, states design goals, provides implementation

PAGE NUMBERING AS ORIGINAL

specifics, studies advantages and disadvantages of proposed cluster scheme, gives experimental results with analysis.

7) Appendices contain fragments of source code which implement most important parts of cluster control software.

CHAPTER 2

LEGACY WEB CACHING TECHNOLOGY

2.1 Introduction to Web Caching

Network traffic is very valuable resource in modern Internet. Monthly cost of long-range communication links often exceeds the cost of equipment used to handle these links, this is especially true for networks outside North America, where traffic cost usually constitutes the biggest part of operational expenses. With explosive growth of Internet traffic in recent few years the task of efficient use of communication links becomes very important. It is widely known that most[6] of Internet traffic is

[6] Until recently more than two thirds of overall Internet traffic was saturated by HTTP traffic. In recent few years P2P networks such as Napster, Morpheus and BitTorrent have eaten a significant part of Internet resources, so at present time share of HTTP traffic is probably slightly lower.

saturated by Hypertext Transfer Protocol (HTTP) used by web servers to deliver static and dynamic content to web clients. Therefore improving efficiency of HTTP traffic will have dramatic impact on the efficiency of the whole Internet and any given network in particular. Internet traffic is likely to grow in the future at fast pace, so efficient use of bandwidth is important today, and it will be even more important tomorrow.

2.1.1 Idea of Web-Caching

There are hundreds of millions of web sites in the Internet, but their popularity differs very significantly. Many sites are accessible via low-speed and/or congested communication link. The idea of web caching is to exploit redundancy in user requests, because users often access the same Internet objects such as news, stock quotes, articles etc, and available bandwidth is wasted by transferring the same objects multiple times. Web cache acts as a proxy between web users and web servers. All users' requests are directed to web cache, that checks whether requested object is present in cache's storage; if the object is present and is not stale, then it is returned to the user without requesting web server of object's origin; if the object is not present in cache's storage, then it is fetched by web-cache, placed into cache's storage and returned to the user (Rabinovich and Spatschak). Virtually all Internet users benefit from the web-caching technology, because web-caching is built into all significant web-browsers, and a few megabytes of disk space at users' computers are allocated to store recently visited pages.

The idea of caching was first implemented in Domain Name System, greatly decreasing load to popular DNS servers (such as root DNS servers or first-level-domain DNS servers). Every DNS record has a predefined time-to-live record, which can be used to cache these records and determine when these records in DNS cache should be freshen from authoritative DNS server. HTTP protocol also has properties which can be used to define caching behaviour such as permission to cache an object and expiration date of object.

Traditional Internet communication model is known as client-server model. Traditional HTTP protocol was built with client-server paradigm in mind. For example, when user requests remote object from web-server, host part of URL is translated to IP address using DNS, and then an HTTP request using method *GET* (or POST) is made to that IP address to TCP port 80 (by default); server handles that request and returns reply to the client. Web cache is a shared network service, which is shared by many users and breaks client-server paradigm by placing an intermediary or proxy between Web users and Web servers. In case of Web-caching when user makes request to remote Web-server, this request is intercepted by Web-cache, which on behalf of Web-user makes requests to original Web-server if object is not found in cache, or object is not fresh. Freshness of given object is determined using either *Expires* HTTP Field or using predefined time-to-live for objects of that type. If object is found in cache and is fresh, then it is returned to the user; this scenario is called *Cache Hit.* When object is not found or is not fresh, then this scenario is called *Cache Miss*, and fetched object is stored in cache alongside with some of its attributes such as Time-To-Live property.

There are two hit rates – object hit rate and size hit rate both usually expressed in percent. These two values are among of the most important parameters numerically describing efficiency of web-cache (alongside with latency distribution). Object hit rate is a number of cache hits divided by total number of users' requests to cache; size hit rate is the size of cache hits divided by the size of all requested objects (taken either from cache or from remote server). Size hit rate roughly describes bandwidth savings achieved by given cache. Typical object hit rates for properly tuned big web-cache range from 40 to 65%, size hit rates range from 20 to 40%. Static objects such as HTML, CSS objects and images contribute most to hit rates, while dynamic pages (CGI programs, ASP or PHP scripting languages) usually can not be cached at all, because developers of these dynamic resources rarely care

about possibility of caching (though it is trivially possible to add reasonable *Expires* header to virtually any dynamic object).

Traditional client-server HTTP has both advantages and disadvantages. Advantages are as follows:

1. Web objects at server can be modified at any time, so web-clients will immediately get new fresh version of content upon requesting.
2. Remote web-server can distinct one user from another (by IP address) and either provide some sort of IP-based authentication (not very reliable though) or produce different content basing on IP address of requesting Web-user.

Main disadvantage of traditional HTTP is that for every request all information must travel across a (sometimes long) network path saturating network bandwidth and increasing the load to origin Web-server. Introduction of Web-caches into traditional HTTP environment leads to decrease of HTTP traffic, somewhat better latencies experienced by end users, lower load to origin web-server and increased risk of delivering stale content. Obviously IP-address-based authentication does not work in case of web-caches because all requests from user are visible at original web-site as requests from web-cache; information in HTTP header about real source IP of given request is usually added by Web-cache and can not be trusted. From the point of view of originating web-server in case of web-cache deployment it is not possible to estimate precisely how many times a specific content has been downloaded from web-server, because many user requests may only access intermediate web-cache, and originating web-server rarely has any control or access to these web-caches. This sometimes causes web-site's owners to effectively disable caching of their pages by setting Expires header to some date in the past; as a result web site gets all requests from end users, none get lost in web-caches, but the effectiveness of the Internet as a whole suffers from these actions. This cache-unfriendly behaviour causes some web-cache operators to ignore *Expires* dates which are older than a few hours in the future and replace them with some predefined value in the future; this is

done to achieve higher hit rates. These kinds of problems are in part caused by deficiencies in current version of HTTP protocol (version 1.1), which was accepted in 1999 and is far from perfect for Web-caching purposes (though much better in this respect than previous version – HTTP/1.0).

HTTP protocol version 1.1 has some mechanisms that help in Web-caching, such as *If-Modified-Since* attribute to method GET, which could help Web-cache to check without unneeded traffic waste whether given object is still valid. If object is valid, then If-Modified-Since returns just confirmation of this fact, without transferring the object itself. If object is not valid, new fresh version of object is returned. If-Modified-Since is primarily used by Web-caches and browsers when expiration date of given object is not explicitly specified. However it is not possible to specify different attributes for particular parts of the page, so the page is always treated as a single monolithic object (thus expiration date of the page will always be the expiration date of shortest-lived fragment of the page).

2.1.2 Web Caching Benefits

Web-caching is a technology which when properly deployed can improve efficiency of bandwidth consumption (so it would be possible to serve *more* requests using the same available bandwidth), improve page retrieval latencies and in some cases reduce network traffic (that is to serve the same number of requests spending *less* traffic). When improperly deployed, however, it can lead to unavailability of some or all web-sites to end users, it can increase latencies (especially in case with multiple levels of web-caching hierarchy and/or underperforming web-caches, not capable of handling load) or cause difficultly diagnosed problems (such as random faults or delays, incorrect handling of time-to-live, so stale objects will be returned by the cache etc.), while requiring significant amount of maintenance effort. From user's point of view the main benefit of web-cache is significantly improved average latency of web-object retrieval (much lower in case of cache hits and slightly higher

in case of cache misses – all that in comparison with direct access). For Internet Service Provider main benefit of web-caching system is either decreased amount of HTTP traffic (due to the cache hits), possibility to serve more users employing the same communication link and/or increased satisfaction of users (due to the lower latencies).

One of the most important benefits of web-caching is bandwidth savings, because significant part of web objects will be fetched from web cache itself thus eliminating need to fetch web object from remote site via expensive (sometimes *very* expensive) communication link. The cost of traffic to and from web cache in this case is insignificant because web caches are usually placed in the same local area network (LAN) or in nearby wide-area network with high-speed relatively cheap traffic. The value of bandwidth savings depends greatly on details of web cache deployment, number of web clients and distribution of their requests. Sometimes this value may exceed 50%. In case of highly congested external links web-caches may not give any bandwidth savings (because any freed bandwidth will be immediately saturated by other requests), but will allow to serve more requests, and therefore leading to more efficient use of bandwidth. Similarly, by decreasing HTTP traffic web caches may provide more bandwidth for other services such as streaming video etc. Even a simplest cache can greatly improve user's experience and improve bandwidth usage efficiency. E.g. in 24 hours after deployment a cache in Cardiff University (Cormack, 1998) achieved a 50% object hit rate.

Quality of service for web clients using web caches is also improved. When requested object is in web cache it is usually delivered to end user much faster than

from original web server. When requested object is not in cache added processing delay is usually non-significant in comparison with the time of object fetching from original server. This can be illustrated on following example. Let us suppose that object size is 8KB[7] and roundtrip time between web-cache and web-user is 10ms and roundtrip time between web-user and origin server is 200ms (quite typical for international transatlantic communication links or even domestic traffic via dialup links). HTTP uses TCP, and minimum complete TCP session requires at least 8 packets coming back and forth. Transferring 8KB will require around 16 TCP data packets (provided MTU is minimum possible; typical LAN MTU of 1500 bytes would require 6 packets) coming from web-server, so the whole transfer would take around 2 seconds. Transfer of the same object from nearby web-cache in case of cache hit would take no more than 100ms (including processing time at the web-cache and the transfer itself), or 20 times faster than fetch of the same object from remote server. This improvement is clearly visible by web-user even without any measuring software. In case of cache miss delivery of web object via web-cache will take 2 seconds (retrieval of object from web-server by web-cache) + delivery from web-cache to web-user and processing time at the web-cache (will not exceed 100ms). Therefore in case of cache miss retrieval of object via web-cache will be at worst case 5% slower than direct fetch from origin web-server, which will be hardly noticed by web-user. In real life this slowdown is even lower, because there is usually present an already established (persistent) connection from user to web-cache, so there is no need to spend time on initiating TCP session. Web-caches are also usually have better connectivity to the Internet, so they generally can fetch requested object faster then their clients via direct connection to origin web-server.

[7] Typical average size of objects in Web-cache storage is about 8-12KB

Therefore in average (provided reasonable hit rates are achieved) overall user experience will be significantly improved by utilizing web-cache.

Web caches may be configured to improve privacy of web users, so any access of web user via web-proxy is visible at remote web server as access from the web-cache itself, so it is not possible to find out what is the IP address of requesting web user. If it is needed web caches may also filter out some HTTP headers containing information affecting privacy. This is especially useful in modern Internet where truly anonymous access to WWW resources is otherwise virtually impossible (using legitimate ways of course). Web caches may be also used to trace and log activity of users. Web-caches usually write all requests' URLs to log files, which can be used later to analyze users' habits and activity, become the proof of certain users' actions or serve as input data in many simulations of improvements to web-caching technology.

Another side-effect benefit of web-caches is its ability to limit access to unwanted content. This feature is especially useful in schools, but sometimes is used in some companies to limit access to entertainment sites and portals to improve productivity of employees. Many web-caches including Squid-based ones support various forms of password-based authorization, so only legitimate users could get access to Internet or Intranet. Some organizations even deny any access to web sites by default, allowing access only to a strictly limited set of web-sites. Similarly web-caches can be used to limit access to commercial parts of some network with paid access. Some intelligent web-caches can rewrite some specific URLs to point to nearby web-servers.

2.1.3 History of Web-Caching

First successful practical implementation of caching in modern Internet was implemented in DNS protocol, and proved a great success. The development of Web caches started in the beginning of 1990s when explosive growth of Internet traffic

caused by World Wide Web popularity led to congestions in Internet communication links not designed from the very beginning to handle big volume of HTTP and FTP traffic. It was shown in 1993 by Danzig that properly located caches could reduce network FTP[8] traffic by 30%. First web caching software (Harvest) was designed around 1993, however its development has stopped in the middle of nineties. Squid[9] is open-source software derived from Harvest. Squid is currently being developed by National Laboratory of Applied Network Research (NLANR). Harvest, and later Squid, founded the basis for various proprietary Web-caching products such as Netapp, BlueCoat etc. Nowadays caching and its applications are used virtually everywhere in Internet including high-performance web-servers, Content Delivery Networks (CDNs), Transparent En-Route Caches (TERCs). Without web-caches congestion problems in Internet would have been much more severe.

2.2 Applications of Web-Caching

2.2.1 Direct Proxies

Direct proxy is the most basic form of web cache. In this case user's web browser is manually configured to forward all users' HTTP requests via web-cache (see Fig. 1). Web-cache fetches object from web site (if it is not already present in cache's

[8] File Transfer Protocol, the primary method of file distribution at that time. Now FTP is quietly fading away due to its firewall unfriendliness, use of two TCP connections, complexity of implementation, impossibility of virtualization (many servers at one IP) and presence of better alternatives such as HTTP or peer-to-peer (P2P) networks such as BitTorrent or Napster.

[9] Origin of the name 'squid' is not known; however authors of squid describe it at http://www.squid-cache.org: "We needed to distinguish this new version [of caching software] from the Harvest cache software. Squid was the code name for initial development, and it stuck."

storage) and then returns it to the user. This approach in general case has significant disadvantages:

Each and every web browser has to be configured manually to use web-proxy. This drawback is especially significant in case with many hundreds of thousands of web users, making direct proxy completely or partially useless. When there are several caches present, load to these web-caches must be distributed either manually, or via proxy autoconfiguration script (Netscape).

It is easily possible for end user to bypass proxy thus eliminating any proxy benefits. Therefore it becomes impossible to enforce policy of mandatory web-cache usage.

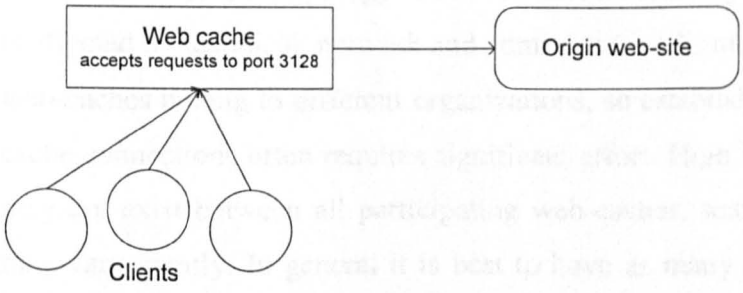


Figure 1. Direct web-cache.

However it is possible to stimulate users to use proxy by limiting direct HTTP traffic (HTTP traffic going via proxy will stay unlimited), so performance degradation when using direct access in comparison with access via proxy will be clearly visible and significant. Users should be made aware of this policy though to prevent increase of support requests. The only way to force all users to use web-cache in case of direct proxy is to block *all* outgoing HTTP traffic to outside network, so the only way to browse Internet would be to use web-cache (this method is usually called *forced caching*). Any attempts of direct access to remote WWW servers will result in timeouts. Still, this method is not perfect, because it causes a big quantity of support requests, and the same result can be achieved by using more flexible technique (transparent caching). So basically direct proxy is suitable for a network with small number of computers (home or small office network), and in such environment is usually easy to setup, configure and maintain.

2.2.2 Cache Hierarchies

Performance of single web-cache is usually limited by many factors. It is sometimes practical to combine several web-caches into a big virtual cache, or *hierarchy* (see Fig. 3). In hierarchy, caches are organized in a tree-like structure with higher-level caches and lower-level caches with web-users residing at the bottom of the tree. Web-caches interoperate with each other using various protocols such as HTTP, HTCP and ICP. Optimal place for higher-level caches is next to the most expensive communication channels (Li et al, 1999).

Choosing the proper topology for a web-cache hierarchy is not a trivial task, which is affected by technical, network and administrative limitations. Sometimes different web-caches belong to different organizations, so establishing and maintaining inter-cache connections often requires significant effort. High speed network connections may not exist between all participating web-caches, scalability of hierarchy nodes may vary greatly. In general it is best to have as many inter-cache connections as required for decreasing latency and maximum hit rate, but no more. There are 2 different relationships between nodes in hierarchy – child-parent and sibling-sibling. In child-parent relationship child asks its parent for an object; if this object exists in parent's cache, object is forwarded to child; if object does not exist in parent's cache, then it is downloaded by parent, stored in its cache and then forwarded to child. Child never requests origin web-servers directly, parent never requests child for a web object. In sibling-sibling relationship both parties have equal rights; if first sibling needs to get an object not present in its cache, first sibling asks second sibling for that web object; if object is present in second sibling's cache, then it is returned to first sibling; if object is not present in second sibling's cache, then first sibling fetches web object directly from origin server (or, in case of multi-level hierarchy, from its parent cache). Even 3-node web-cache hierarchy may be constructed in many ways (Du and Subhlok, 2002) as shown in the following Fig. 2 (not all possible combinations are present).

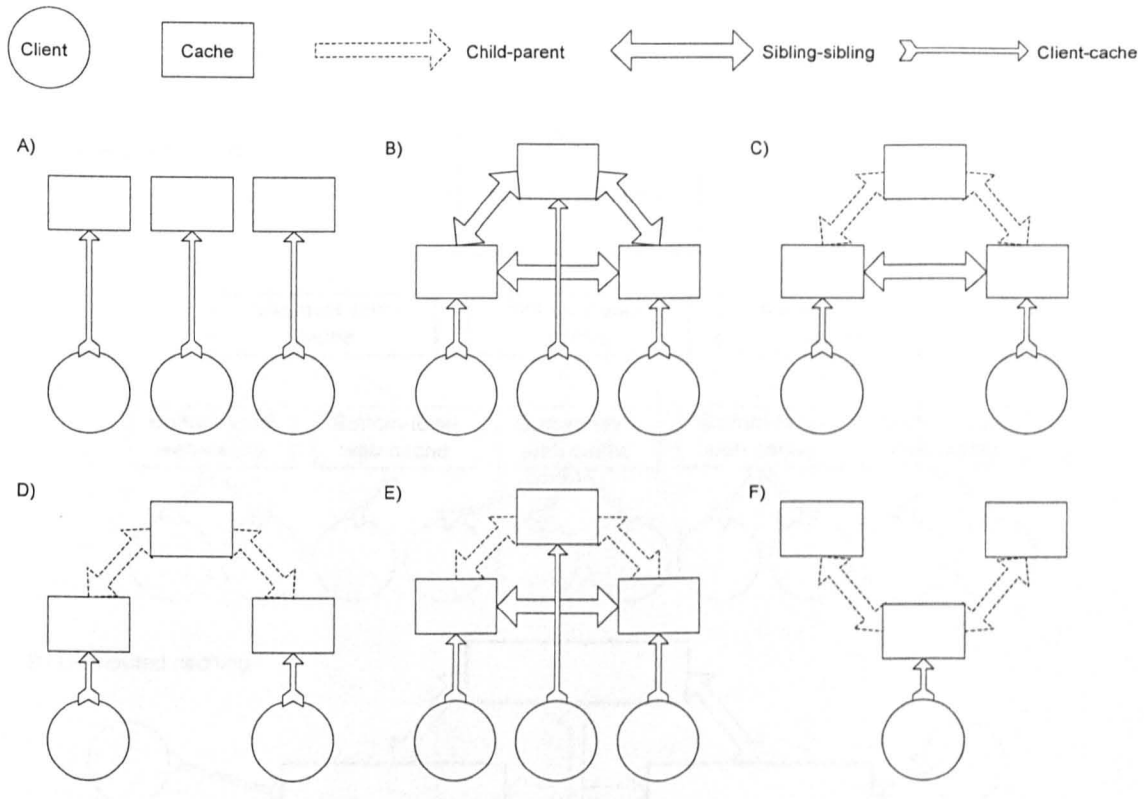


Figure 2. Web-caching hierarchy relationships.

Synthetic benchmark performed by (Du and Subhlok, 2002) et al shows that the best results both in terms of latency and hit ratio are achieved when using B) scheme (with 3 siblings). At the same time tree-like scheme D) (with 1 parent and 2 children) shows rather poor latency and performance. However with the increasing number of nodes in hierarchy it quickly becomes impossible to maintain relationship between any given pair of nodes, because any relationship requires network connection, bandwidth and processing power, so in real world any web-cache hierarchy will be mostly tree-like hierarchy containing local clusters of siblings (which most likely belong to the same institution or company).

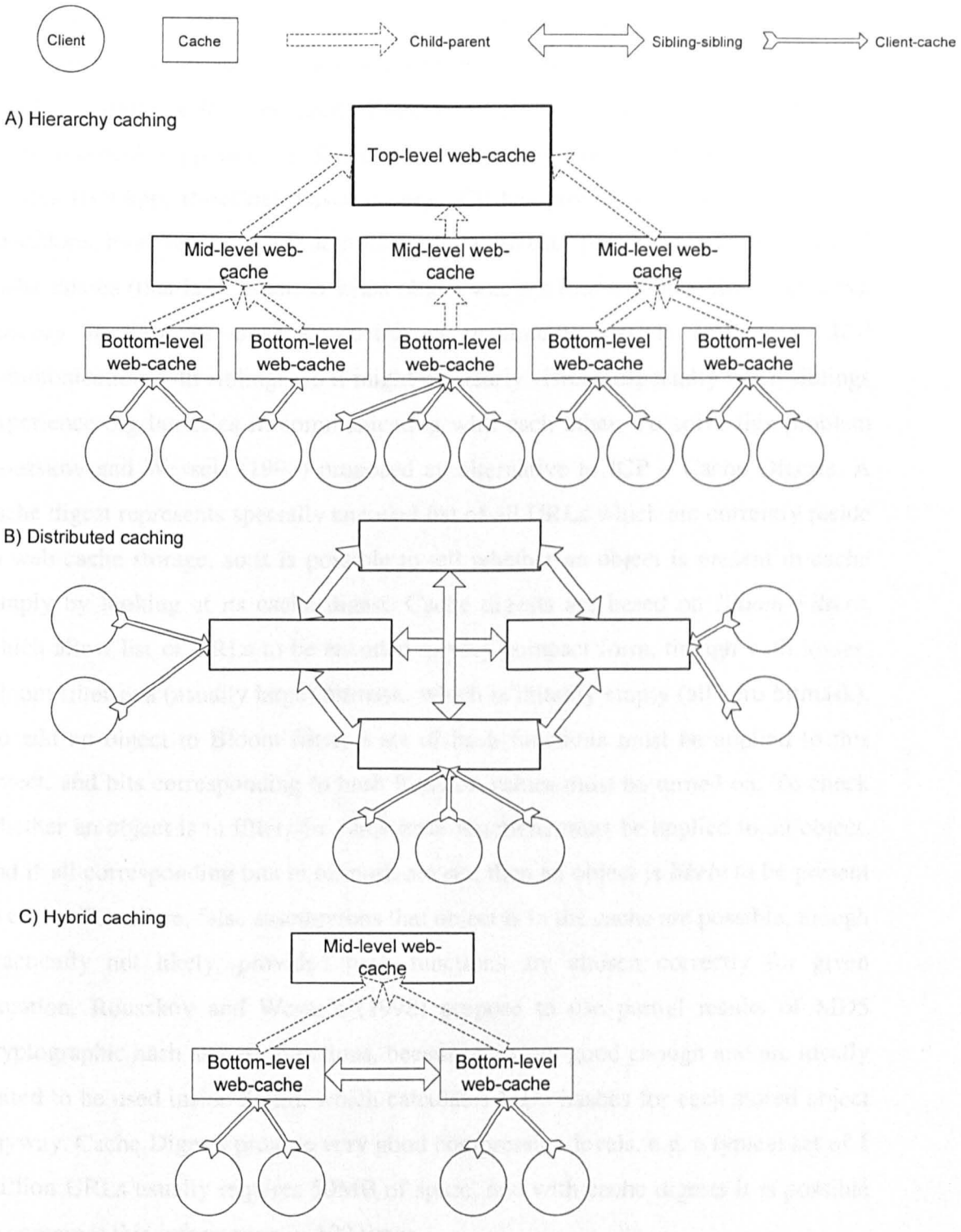


Figure 3. Web-cache hierarchies.

The efficiency of web-cache hierarchy depends on ability of participating web-caches to quickly locate objects stored in siblings' caches. The most common way for this purpose is Internet Cache Protocol (ICP) – a simple, light-weight UDP-based protocol supported by Squid and many proprietary caching software and devices (NetApp, BlueCoat, Cisco cache). ICP has proven its benefits in real life conditions, however it can add significant delays to user-perceived latency in case of cache misses (that is in situation when object was not found in all siblings' caches). Latency in case of miss is usually a maximum possible latency in ICP communication with siblings, so it might be clearly visible, especially when siblings experience big latencies in communicating with each other. To solve this problem Rousskov and Wessels (1998) proposed an alternative to ICP – Cache Digests. A cache digest represents specially encoded list of all URLs which are currently reside in web-cache storage, so it is possible to tell whether an object is present in cache simply by looking at its cache digest. Cache digests are based on *Bloom Filters*, which allow list of URLs to be encoded in very compact form, though with losses. Bloom filter is a (usually large) bitmask, which is initially empty (all zero bitmask). To add an object to Bloom filter, a set of hash functions must be applied to this object, and bits corresponding to hash function values must be turned on. To check whether an object is in filter, the same hash functions must be applied to an object, and if all corresponding bits in bitmask are set, then an object is *likely* to be present in cache. Therefore, false assumptions that object is in the cache are possible, though practically not likely, provided hash functions are chosen correctly for given situation. Rousskov and Wessels (1998) propose to use partial results of MD5 cryptographic hash as hash functions, because they are good enough and are ideally suited to be used inside Squid, which calculates MD5 hashes for each stored object anyway. Cache Digests provide very good compression levels, e.g. a typical set of 1 million URLs usually requires 50MB of space, and with cache digests it is possible to compress this information in 100 times.

Web hierarchies do not use aggregate disk space efficiently, one document might be cached in many caches of hierarchy, and adding another node to hierarchy does not add disk space accordingly. Let us consider a situation with two nodes in parent-sibling relationship. When a requested document is not found in child's cache, the request will be forwarded to parent's cache, fetched by parent from origin web-server, stored in parent's cache, brought back to child and stored in child's cache, and only then forwarded to child's client. Therefore that document will reside in both caches, effectively decreasing aggregate cache size of hierarchy. As shown in study of Ramaswamy and Liu (2002), this duplication decreases both object and size hit rates of hierarchy, in most cases it also increases latency. The consequence of this fact is that it is always better to have fewer caches in hierarchy, but more powerful ones. Ramaswamy et. al. propose a scheme to reduce replication of documents in hierarchy and therefore increase aggregate disk space of hierarchy using a modification to cache document replacement algorithm, which takes into account object average expiration age of every cache.

Hierarchies have several drawbacks (Rodriguez et al, 1999) which should be taken into account when designing web-caching hierarchies. First, every hierarchy level introduces additional delays to request serving time. Second, the higher web-cache is in hierarchy, the more requests it serves. Therefore, if its performance, reliability or scalability is not adequate, this higher-level web-cache may become the bottleneck of the whole hierarchy and even efficiently render it unusable. Third, overall disk space of hierarchy is not always used efficiently, because the same document might be stored at several hierarchy levels.

Another way of cooperation among web-caches is distributed web-caching (see Fig. 3). In this approach all caches have equal rights and serve each others' misses. Distributed web caches cooperate using either ICP protocol or cache digests (Rousskov and Wessels, 1998). When distributed caches are placed at lower parts of network, where links are high-speed and not congested, distributed web-caching can

produce significant benefits such as good hit rates and efficient disk space usage. But the bigger the number of cooperating caches is, the bigger (exponentially) is inter-cache traffic, administrative issues also increase.

User perceived latency consists of two parts – connection time and transmission time. Connection time is a time required to connect to nearby cache and wait for cache to either find the requested document in its disk cache or fetch it from parent cache or from origin web server. Transmission time is a time required by client to fetch document from web-cache. For big objects transmission time is more significant than connection time, for small documents vice versa. What is important for end user however is total time. In comparison with hierarchical caching distributed caching has lower transmission times, but higher connection times (Rodriguez et al, 1999) (because lower-level web-caches of hierarchy are usually located closer to web-user than distributed web-caches). It is possible to combine benefits of both topologies by using a hybrid scheme (Rodriguez et al, 1999), where there is a multi-level hierarchy and nodes at the same level can cooperate with each other.

2.2.3 Reverse Proxies (HTTP-Accelerators)

Most of the web-sites today provide a highly dynamic content. Generation of such content is usually very resource consuming, seemingly simple page may require multiple database queries or other CPU or disk-intensive operations, like a few requests to Java application server etc. Therefore even a modest number of user requests can push web-server and underlying infrastructure such as SQL databases to its limits. Web-caches in accelerator mode (see Fig. 4) allow caching some of the replies and greatly decrease the load to web-server itself. Web-accelerators are usually placed at web-server's location and pass through all users' requests.

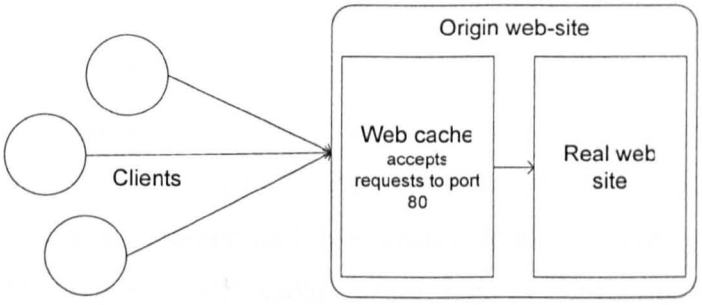


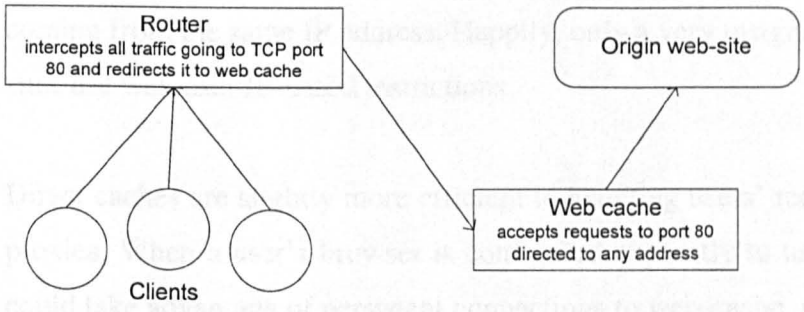
Figure 4. Web-cache in accelerator mode.

Dynamic content is generally considered non-cacheable, because it is not possible to predict expiration dates for such content, and sometimes it is not possible to say that dynamic content is cacheable at all, because multiple queries with absolutely the same parameters can return different results each time. Also, different parts of the page may have different properties, some fragments may have a long time-to-live, and some can not be cached. Web-accelerators are built to cache dynamic content, and they can do it efficiently, because they are tightly integrated with web-server, and they know in advance (it is defined in web-accelerator configuration) which parts of the web-site can be cached, what is TTL for different pages. Some advanced web-accelerators even can combine web-pages from fragments and apply different storage decisions to these fragments.

2.2.4 Transparent Web-Caches

Deploying web-caching hierarchies is a difficult task mostly due to miscellaneous organizational and administrative issues. Achieving high efficiency of web-caches requires that as much as possible of HTTP traffic to be routed through web-caches,

which is difficult [10] or even impossible to achieve in traditional web-caching schemes, because every user's web browser must be configured to use web-cache. Transparent caches are configured in a special way, so they intercept *all* HTTP traffic to well-known TCP port number 80 (see Fig. 5). Classical transparent caching involves a router and web-cache; router intercepts HTTP traffic and directs it unchanged to web-cache. Web-cache is configured to accept connections to any address (in *promiscuous mode*), serve request and return request to the user. Transparent web-caches can achieve maximum possible bandwidth savings, but they also must be highly reliable and must scale well. Failure of transparent web-cache obviously will lead to denial of service for its users, because it is not possible to override transparent proxy. Router which accepts and redirects to web-cache all HTTP traffic usually does not have any means to check whether web-cache is actually working; Web Cache Coordination Protocol (WCCP) allows router to quickly detect and detach failed web-caches, so users will not lose HTTP connectivity in case of web-cache failure.



[10] Forced caching does direct all web traffic via web-cache, but in this case a manual configuration of users' browsers is needed, and when browser is not configured properly it will result in users' frustration and increase of support requests.

Figure 5. Transparent web-cache.

Transparent web-caches is a natural choice for organizations with big number of users, because end user configurations are simply not needed, and therefore administrative problems do not exist. Deploying reliable web-caches requires much of effort, because performance and reliability requirements often exceed capabilities of standard solutions. After transparent cache is deployed it usually requires relatively little effort to maintain and operate.

Transparent caches may seem impossible to detect, but it is not true. Due to the fact that transparent cache "steals" users' identity, IP-based authentication will not work. Also, some sites which limit the number of connections coming from a given IP-address may incorrectly identify a transparent web-cache as abuser and therefore deny all access to all but a few of web-cache's users accessing that site. For example, when two users try to access simultaneously via transparent web-cache a certain web-site allowing only one connection from a single IP, only one of these users will be given access, because both users' connections are visible at that web-site as coming from the same IP address. Happily, only a very insignificant amount of web-sites use web user IP-based restrictions.

Direct caches are slightly more efficient in handling users' requests than transparent proxies. When a user's browser is configured explicitly to use web-cache, browser could take advantage of persistent connections to web-cache, so connection does not need to be re-established every time when working with different web-sites. When browser uses transparent cache, it does not know about existence of web-cache, so to access any site a new connection must be established to remote site which would be rerouted to web-cache. In many cases this factor is not significant, because usually transparent web-caches have very good connectivity with their users. Transparent caches also rely on the presence of complete URL in HTTP headers of every request; happily all reasonably modern browsers do insert such URLs; some

non-standard tools using HTTP may not work with transparent caches if they do not follow this requirement.

2.2.5 Transparent En-Route Web-Caching

Transparent web-caches are generally used in corporate or institutional networks and are located next to Internet connection of these networks. Internet Service Providers use another kind of transparent web-caches – transparent en-route web-caches (TERCs) – which are usually placed on backbone communication links and provide significant bandwidth savings (Krishnan et al, 1999). Basically TERCs combine router and transparent cache in one device. This combination has several advantages. First, very low latencies of serving request. In case when object is stored in cache's memory, it is returned immediately. In case when object is not present in cache, it will be fetched and stored in cache. TERCs also are capable of detecting overload of caching part; in this case TERCs just pass requests through without touching caching part (Johnson).

2.2.6 Web-Cache Evaluation

It is important to measure effectiveness of caching system (Davison, 1999). Principal operating parameters of web-cache include ability to handle a certain traffic in megabits and/or handle a certain number of requests per second (these two parameters are basically the same, because average object size in web-cache storage is nearly constant in the range from 8KB to 12KB); other important parameters include object hit rates (a number of cache hits divided by total number of requests) and size hit rates (size of hits divided by size of all requests).

One of the most widely used ways to analyze miscellaneous operating parameters of web-caching system is to study log files of web-caching software (Barford and Crovella, 1999). Analyzing web proxy log files is not a trivial task. Web-cache logs only contain partial information which is often not enough for web-cache effectiveness' estimation. Most of HTTP headers which are crucial for

understanding web-cache behaviour are usually not recorded to keep the size of log files in reasonable limits. To collect web-cache logs suitable for analysis a special kind of web-proxy is required – cache-busting proxy (Kelly, 2001). The cache-busting proxy effectively disables end-user caching (e.g. in Microsoft Internet Explorer) by modifying response headers from web-servers, so that these responses will not be cached locally; this helps to estimate which part of web objects may be cached at all.

Kelly (2001), who studied a massive multigigabyte anonymized client trace of WebTV customers, describes a perfect web caching system in terms of maximum hit rate, however this system is non-practical, because it requires storing of *all* web objects for an extended period of time (in boundary case - forever) and also needs significant changes in HTTP protocol, web browsers and web caches. According to Kelly (2001), browser-to-cache interaction starts with an ordinary HTTP request; web-cache answers to client with checksum (e.g. MD5 checksum), and client compares this checksum to the checksum of locally stored object (which are stored for a *long* time), if checksum is the same, then object is not fetched. However, this approach can not be deployed throughout the Internet, and is limited to proprietary networks, where *both* browsers and web-caches are controlled by the same entity, such as WebTV.

Another approach to analyze operation of any given cache is to use a special web-caching benchmark (Polygraph; Wisconsin; Du and Subhlok, 2002; Bent and Voelker, 2002). These benchmarks can make almost any particular workload to web-cache, allowing to estimate performance limits of web-cache or to study effectiveness of its part. Unfortunately these benchmarks do not use real-life web-pages consisting of properly grouped objects, which may affect benchmark results for some applications.

Yet another approach to web-cache evaluation would be to use a specialized web-caching software, which either has extended logging and reporting capabilities at the expense of performance, or has other non-standard capabilities, not suitable for general-purpose caching (Koletsou and Voelker, 2001; Kelly, 2001).

2.2.7 Content Delivery Networks (CDNs)

Content Delivery Networks (CDNs) is probably the most important application of Web-caching nowadays. CDN can be defined as a set of large number of distributed caches located at the edges of network, which are used to push valuable content closer to the end-user thus reducing network traffic and required bandwidth. Intelligent DNS implementation is used to direct user to the nearest cache (see Fig. 6). CDNs are deployed transparently for end users; they provide very good traffic savings and are widely used in modern Internet. Needless to say, CDNs require very scalable fault-tolerant web-caches, thus being the ideal application for web-caching cluster. Many approaches are proposed to make CDNs more effective. One of the approaches (Kelly, 2001; Panteleenko and Freeh, 2001) is to make web-servers more effective, thus increasing complexity of already complex system. These optimizations, sometimes, micro-optimizations might provide some performance boost, but they can never improve performance by several times.

In perfect world, where web-caches are deployed ubiquitously, CDNs delivering static content are not needed (Rabinovich et al, 2003), because these ubiquitous web-caches will act as a world-wide CDN. However we live in a real world, so even simplest CDNs might provide great benefits when implemented properly. CDNs are now used not only for serving static content (HTML pages, images, CSS etc.), but also for video streaming services (Merwe et al, 2003) and more complex applications (Rabinovich et al, 2003).

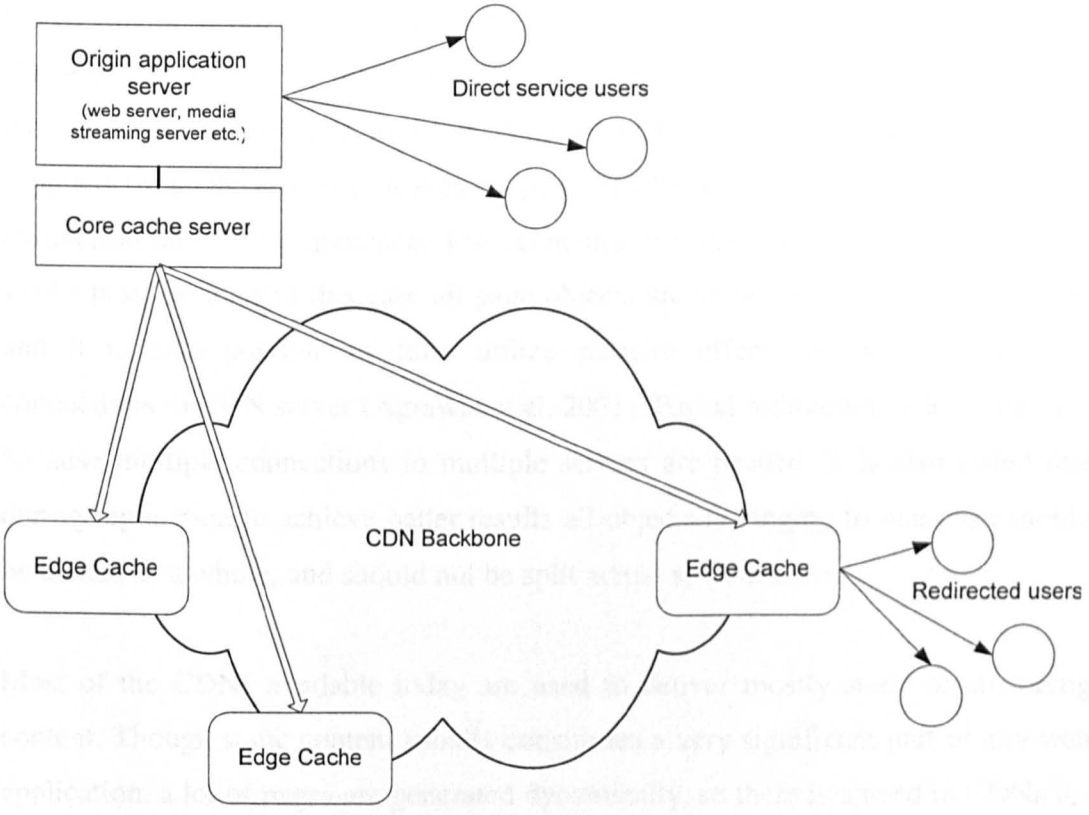


Figure 6. CDN Topology.

Content Distribution Networks use redirection to provide best possible experience for its users (Davison). Redirection can be accomplished either by means of HTTP protocol or via DNS. CDN redirection via HTTP is not widespread, because it requires from origin web server to handle this request, which may be time-consuming due to the network congestion or distance between client and web server, and it also produces unnecessary load to origin web server. When DNS redirection is used, a user requests an IP address of a web-site using DNS, then a specially modified DNS server looks at user's origin IP and gives the IP address of the CDN server, best suitable for a given user (ESI). All further requests to the original web-site will be served by the nearest CDN edge server. It is obvious that in this redirection scheme a CDN operator must have a full control on web-site DNS zone, which is often undesirable. In that case another approach may be used, when most of

the content is served from original web-server, but some embedded objects (usually big or complex objects) are pointing to a particular CDN server, and redirection to the best CDN server is performed at origin web-site using a small piece of software supplied by CDN provider. Kangasharju et al (2000) have studied the effects of redirection on user's experience. They conclude that complete redirection via DNS works best, because in this case all page objects are served from the nearest server and it is also possible to fully utilize positive effects of persistent parallel connections to CDN server (Agrawal et al, 2001). Partial redirection is less effective, because multiple connections to multiple servers are needed. It is also stated that during replication to achieve better results all objects belonging to one page should be treated as a whole, and should not be split across several servers.

Most of the CDNs available today are used to deliver mostly static or streaming content. Though static content usually constitutes a very significant part of any web application, a lot of pages are generated dynamically, so there is a need in CDNs for dynamic content, or CDNs for applications (Rabinovich et al, 2003). Application CDNs (ACDNs) provide automatic migration of applications across network servers, load balancing, application consistency management. ACDNs are the most promising way to solve web scalability problems with a positive side effect of decreasing content delivery latency for end users and decreasing possibility of network congestion.

2.3 Squid Web-Caching Software

Squid is probably the most widely distributed and used web-caching software. It is open-source and runs on virtually any Unix platform (including FreeBSD and Linux). It was derived from ARPA-funded Harvest software, which was the first web-caching software available. The origin of "Squid" name is better explained by one of its authors, Harris Lament – "All the good ones were taken. We needed to distinguish this new version from the Harvest cache software. Squid was the code name for initial development and it stuck". Squid is currently supported by National

Laboratory for Applied Network Research (NLANR), and Duane Wessels is its lead developer. Numerous contributors around the world provide testing and improvements to Squid.

Squid is a high-performance single-process application which uses non-blocking asynchronous IO and a set of external processes to perform DNS lookups (dnsserver processes), rewrite URLs in users' requests (redirectors in Squid's terminology) or (optionally) perform requests to non-standard storage. Squid supports caching of HTTP, FTP and SSL protocols, various inter-cache communication protocols and methods such as ICP, HTCP, WCCP and Cache Digests, and can be used to build very flexible hierarchies or meshes of caches with arbitrary topology. Squid is extremely configurable, virtually any run-time parameter is well documented and could be tweaked if such a need exists (after all, source code is always available); it has extensive Access Control Lists (ACLs), provides support for running in direct mode, web-accelerator mode and transparent mode; supplies HTTP header filtering capabilities, which can be used to improve privacy by hiding some information from users' requests; provides SNMP access to its statistics and configuration; has a simple Web-interface, which can be used to monitor cache's behaviour and perform various actions; supports authentication from many sources; and has good logging capabilities, allowing to produce precious traces of users' activity.

Squid uses resources efficiently, can take advantage of multiple hard drives by distributing stored objects evenly across these disks, so RAID array is simply not needed (and sometimes even hurts – e.g. RAID-5 with 4 disks will have lower capacity – by 1/3 – and perform badly for write operations in comparison with the bunch of 4 disks of equal size). Squid supports caching of objects in RAM, so most popular objects are fetched without any disk access whatsoever (provided there is enough RAM of course; for low-memory web-caches this feature may be disabled if desired). Squid uses RAM wisely by storing only that information which is needed to quickly locate objects in persistent storage (hard drives).

Squid has proven itself as reasonable fast and reliable application with a wealth of features. It is ideal candidate for almost any research project in the field of web-caching. By coincidence it is also a web-caching application of choice for most ISPs. Thus no wonder it was used as the integral part of building blocks for proposed web-caching cluster described in this thesis.

2.4 Summary

Web-caching is a mature technology, which exploits significant redundancy of HTTP traffic and provides bandwidth savings and faster web object retrieval for web-users. Multiple web-caches can be organised into multi-level hierarchies or can form a distributed cache. Web-caching technology in the form of web-accelerators is also used in building high-volume web-sites to solve performance problems of sites' backend. Content Delivery Networks push content closer to the web-user and not only decrease latencies of object retrieval for web-user, but also benefit the whole Internet by decreasing HTTP traffic going via Internet backbone networks. Transparent web-caching allows achieving maximum web-caching efficiency by directing all HTTP traffic to Web-cache; any special setup of users' computers in this case is not required. Web-caching technology in the form of Transparent En-Route Caches (TERCs) enables Internet Service Providers to decrease load on their backbone communication links.

CHAPTER 3

RECENT TRENDS IN WEB-CACHING DEVELOPMENT

Research in the field of web-caching is being conducted in several directions. One of the main directions is decreasing average latency of object fetching for end users, which is the only visible benefit for them. Optimizing bandwidth consumption is a benefit visible to the whole company or organization, this goal may be achieved by improving web-caching software, HTTP protocol or web-server's information presentation. Improvements are proposed in handling most traffic consuming resources such as streaming media. Efficient caching of dynamic content is another direction of research. Designing web-caches capable to handle big traffic is a field of study for many researchers.

3.1 Decreasing Latency

Faster web-object retrieval (that is decreased average latency) is the only benefit of web-caches visible to end users. Decreasing latency will have positive impact on user's browsing experience and will improve their productivity. HTTP has never had decreasing page latency retrieval as a design goal, nor has it had web-cache friendliness. Latency of loading the complex whole page consists of loading an HTML page, parsing it and then loading objects embedded into page, each possibly requiring a separate TCP connection. Multiple and persistent connections partly help in solving this design defect, but anyway the page is never treated as the same object. It is also not possible to specify metadata for the different parts of the HTML page, for example, page time-to-live will either not be specified (most frequently observed behaviour) or set to the TTL of the shortest lived page's fragment. For more

efficient content delivery and caching it is needed to use bundles of objects (Chi et al, 2003), so it would be able to fragment bigger objects without any negative effects of traditional Web model. These bundles must support web-caching (Chi et al, 2003), so it would be possible to cache bundles' parts; in case when one of the parts of bundle becomes stale, cache would be able to fetch them from web-server and repackage the bundle (this may even include re-creation of HTML page from parts). Bundles of objects significantly reduce the number of freshness check (because for every page only one such request would be enough, but not up to hundreds requests as it is at present time), they also increase hit rates, because non-changed parts of the HTML pages and other objects will not have to be retransmitted. Also, transmission of one large object is usually faster and more bandwidth-efficient than sequential transmission of its parts. Practical implementation of bundles will require modifications to web-servers, web-caches and web-browsers, though the bundles are not likely to appear in near future.

Web-server latency can be improved by dividing users into classes, so more resources will be allocated to high-priority users, so quality of service will be higher for these users. This approach is used by some download sites, which have two classes of users – ordinary users who do not pay for downloading and premium users who pay some monthly fee and have lower latencies and more bandwidth available. Particular details of implementing this approach are described in study of Kanodia and Knightly (2003). Improvement of quality of service for one class of users in this approach is gained at the expense of other users; so this approach can be used only in very limited set of cases. Most of web-caching solutions do not differentiate users, and fair division of resources among users is generally implemented.

Latency in downloading web object is important factor, but users are interested in whole pages, not separate objects; and typical page consists of several objects, such as images, HTML documents, Cascading Style Sheets (CSS); some pages may even

consist of hundreds of web objects. Bent and Voelker (2002) study effects of web browser behaviour on overall page download time. They state, that persistent connections (connections which can be used to fetch several objects, so there is no need to spend time on establishing TCP session) and parallel downloading (usually up to 5 connections, for example, page's images may be downloaded in parallel) of page may dramatically affect download times of whole page; and the more complex is the page, the bigger is the impact of advanced downloading techniques. Happily, modern browsers like Internet Explorer and Netscape Navigator do support these techniques, so it is crucial for any web-cache software to provide persistent connections and parallel downloading features.

Decreasing average latency of web objects retrieval is one of the web-cache's benefits. Koletsou and Voelker (2001) researched latency using their own tool – Medusa Proxy, which acts as non-caching forwarding proxy with parallel retrieval and extended logging capability. They have evaluated several cases – web-cache hierarchy, CDN and one boundary case – a special kind of web accelerator (as they call it – Medusa Proxy). In all cases all interaction of user with remote web-servers was going through Medusa Proxy and then either via web caching hierarchy or CDN or directly to origin web-server. In case of web-cache hierarchy they studied NLANR web-caching hierarchy; they were using one of the NLANR web-cache servers that was located very close (in the same local network) to Medusa Proxy. The conclusion was that 63% of HTTP requests were served from cache at least as fast as from origin web-server; mean latency was decreased by 15%. This experiment gives a rough upper estimate of latency benefit given by web-cache hierarchy, because it was performed in almost ideal situation, where connection to the Internet *and* web-cache is never congested. It is worth to note, that 15% is rather significant improvement, as explained below. For fast sites connection through proxy will likely be the same or slower than direct connection, though this relative slowness will not be noticed by the end user, because absolute retrieval times will be low. For slow sites, which can serve a page in several seconds at the best,

improvement in latency when working through web-cache may be dramatic (e.g. less than 1 second instead of several seconds), and this difference *will* be visible to the user, greatly enhancing user's web experience.

As example of CDN Koletsou and Voelker (2001) used well-known Akamai system. By Akamai's own estimates, its CDN handles 15% of overall HTTP Internet traffic, so Akamai has a significant impact on the Internet as a whole, and is the ideal example of CDN for study. Akamai's CDN only improves availability for its customers' web servers and does nothing for all other ones. As stated by Koletsou and Voelker (2001), Akamai edge servers are able to handle HTTP requests in average 5-6 times faster than origin web-servers, proving Akamai's effectiveness. Taking into account all sites however (both Akamai's and non-Akamai's), overall improvement in mean latency was only 2%.

Koletsou and Voelker (2001) also studied a special case, when their only goal was to minimize latency at any cost. In that case they were running their Medusa software in parallel retrieval mode, so when user was asking Medusa server for an object, Medusa server was requesting that object simultaneously from multiple sources, including NLANR web-cache and origin-web server, and returning the fastest response. Obviously, in this case bandwidth was not spent wisely, with object retrieval requiring downloading two or several times the size of an object. Mean latency was improved by 38% comparing with using just NLANR web-cache (therefore, in comparison with direct object retrieval mean latency improvement was even higher).

Overall, Koletsou and Voelker (2001) have shown, that CDNs and web-cache hierarchies can really (and not only theoretically) improve latency for end user.

Some objects are fetched from web-cache slower than from origin web-server. These objects include resources not yet present in web-cache, secured SSL

transactions and dynamic objects, which are usually not cached. Chen and Zhang, (2002) propose a minor modification to web browser which would bypass web-cache for objects which are known in advance (based on URL analysis) as non-cacheable. This modification is easy to implement, because it does not require any changes to HTTP protocol nor to web-caching software. Taking into account a significant number (around 17% for real IRCache web-cache trace used by Chen and Zhang) of dynamic documents (this includes CGI programs, ASP and PHP scripting languages, Server Side Includes), such modification may reduce object retrieval latencies, improve security of encrypted transactions and decrease load on web-cache.

3.2 Caching Dynamic Content

Caching dynamic content is not a trivial task. Time-to-live of dynamically generated objects can not be known in advance, may depend on parameters used to generate this dynamic content; dynamically generated page may consist of several fragments each having a different TTL. To efficiently support caching of dynamic content both Web application modification and web-caching software modification are required as shown by Yuan et al (2003).

Increasing number of dynamic pages negatively affects performance of web-caches (Brewington and Cybenko, 2000; Challenger et al, 1999). But web-servers themselves also suffer from increasing complexity of web-pages. Some pages require many seconds to generate, and a very complex database queries may be needed to produce seemingly simple page. This also causes increased latencies for web-users, and requires very significant expenses to scale web and application servers or introduce additional servers. To address this issue web-accelerators were proposed (Feenan et al, 2002), which sometimes are called reverse proxies. Direct web-caches usually serve requests of users to numerous web-sites; reverse proxies serve requests to just one or few sites, residing near original web-site. The main purpose of reverse proxy is to ease the load to original web-site. To achieve that goal

the reverse proxy must have at least some knowledge about the nature of application running at the original web-server, so dynamic content could be cached effectively. Dynamic content is often unpredictable and often has a very short time-to-live, so usually reverse proxies are placed at web-server's location and sometimes can even construct pages from fragments (for example, Oracle9iAS Web Cache has this functionality), taking into account different metadata of various fragments (e.g. some parts of the page may have different expiration times). Since reverse proxy is located so close to original web-server, it is often invisible by end users, and the whole web-site consisting of web/application server(s), database(s) and reverse proxies looks like a single entity. Web accelerators are placed in critical data path (that is, all users' requests go through them), so they must meet the same requirements as applied to transparent web-caches.

Virtually every web page consists of fragments with different properties such as expiration time, author, origin etc. There are numerous proposals to add support of object bundles to HTTP protocol. Orman (2001) proposes yet another way of specifying object bundle with possibility to ensure integrity of web page's fragments by using cryptographically strong electronic signatures for each fragment. In accordance with Orman's proposal, the whole page is described by a *manifest* – a file containing properties of every fragment of the page written in special language; these properties include origin, author, and modification permissions. Manifest file allows for intermediate systems such as web caches to cache various fragments differently, or change or even delete some parts of the page, and the correctness of intermediate systems' actions can be always validated by downloading manifest file from origin server and comparing signatures and integrity of every fragment and the whole page. Checking digital signatures however is quite computationally expensive process, so proposed solution should be used only for sensitive data.

Datta et al (2002) note, that whole cached dynamic pages have only a limited (and often unknown) reusability, which causes low hit rates. At the same time when web-proxy only considers URL when making decisions, it may cause undesired behaviour of web-cache, because a dynamic page may produce different output for two requests with identical parameters depending on cookie or authorized user's name or IP address of the web-user. It is also stated that dynamic pages corresponding to the same URL may differ both in data and in data layout, and these should be treated separately. E.g. many sites allow users to customize site's presentation, at the same time the data on the page will be the same. Datta et. al. study benefits of dynamic pages' caching using web-accelerator, where presentation layer is separated from data layer and web-pages are treated as a bundle of objects. They conclude that web-accelerators for better efficiency must take into account layout of pages as well. Obviously to achieve that web-accelerators must know the structure of the site and must have access to its inner parts. The technique of separating data from layout is not new and has been proven to be successful at many high-volume sites such as mail.ru. Universal web-caches suitable for handling arbitrary sites with dynamic content can not be built in this way.

3.3 Improving Hit Rates

Web-prefetching is quite an old approach to improve hit rates of web-caches by fetching web-objects which are *likely* to be requested in the future. The effectiveness of this approach is questionable, and HTTP does not have proper support for this technique (Davison, 2001). The main problem of web-prefetching is impossibility to predict which objects will be requested multiple times in the future and possible bandwidth losses caused by downloading unpopular objects. Web-prefetching of popular pages does not have much sense either, because without web-prefetching these pages will be cached after first request anyway, so web-prefetching of this page will save just once the size of the page. Web-prefetching is suitable only in very specific conditions, e.g. when bandwidth is strictly limited and it is known in advance that some specific pages will be downloaded with high probability, so it is

possible to prefetch these pages in off-peak time (at night). Though some commercial web-caches like BlueCoat and Netapp appliances and web browsers including Internet Explorer and Netscape Navigator implement prefetching, it is rarely used and not likely to be developed in the future. Content Delivery Networks (CDNs) and transparent web-caching are more effective and proven ways to save bandwidth. Any potential improvements to prefetching require changes to HTTP protocol (Davison, 2001; HTTP 1.1) and at server side, so these changes are not practical and not likely to be done in any foreseeable future.

Jin et al (2002) propose a way of caching of streaming content using bandwidth-aware caching decision algorithms, with support of partial caching and simultaneous prefetching from multiple sources if needed. Congestion on the path from cache to client is taken into account when making caching decisions. This approach is not applicable however to caching of small objects such as HTML pages, images, cascading stylesheets etc.

In general, caching dynamic content is probably the most promising destination of hit rates' improvement, because around 50% of all HTTP traffic is currently a dynamic content, and this share is likely to increase in the future.

3.4 Summary

The research in web-caching technology is in general oriented on improving two things – latency and hit rate. Latency somewhat depends on hit rate (because object from cache would be served faster than from original web-server), but the former can be improved in other ways. Primary research directions to decrease average latency include changes to HTTP protocol (to improve granularity of web-pages, support bundles of objects) and proposals to enhance existing Content Delivery Networks (CDNs). The most promising way to increase hit rates is to improve hit rates of dynamic content (this value is currently around 0) by implementing changes to HTTP protocol to allow more fine-grained control over page parts (different

properties for each part, for example, time-to-live). A separate task is improving hit-rates of multimedia content such as streaming video; this task is best solved using specially designed CDNs. Web-prefetching, which appeared at least 10 years ago is still to show its practical benefits.

CHAPTER 4

PROBLEMS OF BUILDING HIGH-PERFORMANCE WEB-CACHES

High-performance web-caches are needed when deploying transparent caching for a big network, for top-level nodes in web-caching hierarchies. Existing hardware and software have limitations which must have taken into account when designing large-scale web-caching solutions (Gadde, 1997). Potential problems range from strictly administrative to pure technical ones.

4.1 Problems in Building Efficient Web-Caching Systems

4.1.1 Administrative Problems

Deployment and operation of web-caching systems require significant resources. To achieve maximum benefit from web cache *most* (ideally - *all*) of the users must use that web cache. When using direct proxy or direct access to proxy hierarchy it is almost impossible to assure that every user's browser is configured to properly use web cache (especially when users' computers are out of reach of system administrators, situation which is common for Internet Service Providers). With direct proxies it is possible to completely disable direct access to the Internet by passing web-cache, so all users will *have to* use web-cache in order to get access to Web resources; for corporate network of decent size or for big enough ISP this approach will cause a lot of problems, because it will be required not only to explain to each and every user that direct access is forbidden, but also explain *why* it is forbidden, and how to setup users' browsers properly; obviously, this will cause too

many requests to support personnel. It is also possible to limit bandwidth of allowed direct traffic, so users going directly will have noticeable "slow" connection, but users going via web-cache will have "fast" connection; this approach will not work either for a big organization, because there is still a need to explain "why" and "how", but some users will still use a slow direct connection, so their performance will suffer. Manual configuration of users' browsers takes a significant amount of time, and does not guarantee that user himself or herself would not turn off proxy access off intentionally (by "power users") or unintentionally. So called "automatic" web-cache configuration (Netscape) slightly improves task of browser's configuration, but does not solve this problem. When using direct access it is usually not practical to disable direct access to HTTP servers (to force use of web-proxy), because it leads to significant increase of technical support requests. Allowing direct access leads to possibility to bypass web-cache which is not desirable.

The only way to get rid of administrative problems is to deploy transparent web caching, which does not require any special configuration of users' browsers whatsoever, and a minimal number of support requests from users. Transparent caching also guarantees that nobody will be able to bypass web-cache. Transparent caching requires however reliably working web-cache, because users will not be able to use direct access to the Internet when cache fails. Transparent caching has a few minor drawbacks, such as inability to use IP-based authentication and inability to distinguish at web-server one user of transparent proxy from another one. However transparent caches deliver faster average fetch times and offer significant bandwidth savings (especially true for expensive international communication links).

4.1.2 Technical Problems

A high-performance caching system must satisfy the following requirements:

1. It must provide good enough performance. It basically means that the average latency of user's requests going through web-cache must be lower than, or equal to, the average latency of direct access. This goal is achieved

by deploying appropriate resources such as caching servers and/or optimizing software and configuration parameters of these servers.

2. It must produce bandwidth savings, which usually produce financial savings. This task is solved by allocating enough storage and using efficient caching algorithms.
3. It must be highly reliable, because in many situations there is no possibility for end users to bypass web-cache.
4. Web-caching system must require little effort to deploy and operate.

To meet these requirements, a number of technical problems must be solved. Scalability is probably the most important one. The performance of a single caching unit is rather limited; increasing it beyond certain limit is still possible, but not cost effective. Reliability of single unit is considered inadequate, especially when cheap relatively unreliable PC hardware is used, which usually does not have any redundancy features, such as redundant power supplies and network cards, ECC RAM or RAID disk systems with hot swap support. The most cost effective way of solving these problems is to combine several web-caching units into one big virtual one. There are several methods to accomplish this. The easier one is to use several separate caching units with load balancing at the router. This solution solves scalability problems, but reliability over single caching unit is not improved, and overall hit rate is not likely to be high. Another approach is to use a distributed cache – a set of cooperating web-caches. This method produces good hit rates, but works well only within a certain number of participating web-caches, because inter-cache traffic such as ICP grows exponentially with the number of nodes. Reliability problems are still present in this approach. The only viable cost-effective solution to scalability and performance problems is to implement some form of web-caching cluster with failure detection and load balancing.

4.2 Web-Caching Performance Requirements

Every web-cache has its performance limits. The most significant characteristic of web proxy performance is how many megabits of traffic it can handle. This value depends on what hardware and software are used, how much RAM and disks are used for web-cache. Choosing various parameters of web-cache such as CPU power, RAM size, disk cache size and number of disks is based on general assumptions and practical experience. Danzig (1998) gives good estimates of resources needed to handle every megabit of HTTP traffic. These estimates, though based on outdated hardware data and software characteristics, are still applicable to today's situation with some corrections as follows.

Assuming a 8KB[11] average object size in disk cache, almost every disk write operation will require 2 disk operations – one for file contents and one for file metadata. Most of the modern disks (both IDE and SCSI) support write operations with size up to 128KB, so for the majority of files only 1 write operation will be required (not counting metadata). Advanced file systems such as FFS with softupdates of journalled filesystems such as EXT3FS or ReiserFS may need less than 1 write operation for file metadata write, because metadata writes are delayed and can be aggregated in batches, decreasing number of required writes. Being conservative it is safe to estimate that every write and read operation will require 1.5-2 disk operations in average. It is preferable not to push the disks to its limits, so we will assume in further argumentation that disks operate at 50% disk operations per second capacity. Modern SCSI disks can handle up to 500 random access operations per second, IDE disks handle up to 250 operations per second. This is

[11] This is a conservative estimate; average object size in RUNNet web-caches was around 12KB.

achieved by high rotation per speed of disk plates (typical value for SCSI disks is 10000 RPM and 15000 RPM for the fastest ones), precise mechanics and intelligent controllers. Virtually all SCSI disks and recent advanced IDE disks support *tagged queuing* technique, so multiple write operations in disk's queue can be rearranged by disk controller to achieve higher performance (so two write operations in the same part of the disk could be combined into one actual write operation). As a result we conclude that SCSI disk can handle roughly 250 average object read or writes per second, which corresponds (250 ops/sec is at 50% utilization; for each file read or write operation 2 physical disk operations required, because metadata have to be read/written) to 4Mbps of traffic for every SCSI disk and 2Mbps of traffic for every IDE disk[12], and this is rather a conservative estimate. SCSI disks are better than IDE ones, but the latter are more cost-effective. Realistic estimate would be two times bigger, so resulting rule is to deploy 1 IDE disk per every 4 megabits of cacheable traffic. Not all traffic is cacheable however, a significant part of it will be passed through web-cache without storing it on disk. It is reasonable to assume that only half of traffic is cacheable due to the highly dynamic nature of modern Web sites. Therefore the ultimate rule would be deploy 1 IDE disk per every 8 megabits of traffic.

Required disk space is based on a desire to store several days of external HTTP traffic. Danzig (1998) recommends storing a week of external traffic, but in modern Internet a lot of objects have a short time-to-live, so it does not make any sense to try to store more than 1 day of total HTTP traffic (half of it is assumed as non-

[12] In estimates it is assumed that every object has to be written *and* read from the disk, RAM caching is not taken into account.

cacheable, so this effectively means storing of 2 days of cacheable traffic). A typical size of disks today is 40-200GB for cheap IDE drives. A 40GB drive can store a daily traffic of 8Mbps link at 50% utilization[13]. So the general rule for web-cache size is to deploy 1 40GB IDE disk per every 8 megabits of traffic.

Memory sizing estimate is based on the fact that Web-cache has to keep some information in memory about every object which is cached on disk. Plus some extra memory is needed for web-caching application itself, operating system, network and disk buffers etc. Squid caching software for 32-bit architecture (e.g. IA-32) requires 72 bytes of memory (64-bit architectures such as Alpha or AMD-64 require 104 bytes) for every stored object. For 1GB of disk storage with 8KB (rather conservative estimate) average object size, exactly 9MB[14] of RAM is required. Squid documentation recommends minimum 10MB of RAM for every GB stored on disk. Twice as much is an optimal configuration. This means 400MB of RAM for every 8Mbps of traffic is a minimum, and therefore the rule is 512MB of RAM for every 8Mbps of traffic.

CPU requirements for web-caching application are negligible. Since Squid web-caching software is a single-process single-threaded application, it does not benefit from multiple CPUs nor from hyperthreaded technology present in some Intel CPUs. From a practical experience single 2GHz Intel Pentium IV or AMD Athlon CPU is capable of handling 32Mbps of traffic.

---

[13] This is a correct assumption, because 100% utilization is very rarely met in real life, typical weekly utilization is usually well below 50%. In fact, required space at 50% utilization will be even lower, because IP, TCP and HTTP packet headers have not been taken into account.

[14] 9MB = 72B · (1GB/8KB)

In general, requirements to CPU, RAM and disks for web-caching application which are described above correlate well with Squid authors' recommendations and practical experience of using Squid-based web-caches on IA-32 architecture.

4.3 Limits of Hardware, General-Purpose OS and Caching Software

Every web-cache must store cached documents, and there are two principal storage types – temporary (usually RAM) and persistent (typically disks or disk arrays). Temporary storage is usually quite expensive, but provides excellent random and sequential access times and throughput rates. For temporary storage the difference in sequential access and random access is virtually non-existent. However due to the high price tag of temporary storage it is not practical to store all cache contents inside temporary storage, and second type of storage is needed. Persistent storage is much cheaper per gigabyte in comparison with temporary storage, it provides excellent capacity and acceptable throughput for sequential read and write operations, but the number of random access operations for disk-based persistent storage is low (Maltzahn et al, 1999; Markatos et al, 1999), typically peaking at about 100-400 block[15] transfers per second (the amount of random access operations can be calculated from average access time, which is in the range 2-10 milliseconds). The reason is simple – disks are mechanical devices, so their performance depends on disk's rotations per second, and speed of magnetic head movement, and these values can not be increased with the same speed as CPU performance or RAM size

[15] Block size in this case is 64-128 kilobytes.

grows. Another type of persistent storage – flash memory – is very expensive at present time (it is even more expensive than RAM per megabyte), and this is not likely to change in near future, so the disks are and will remain the only cost-efficient solution for web-cache persistent storage. As shown in (Lee and Tomlinson, 1999; Danzig, 1998), disks are the major bottleneck of every web-caching system. The situation with disk performance is bad, and it will only get worse in the future, because the performance of CPU and RAM sizes grow at fast pace (in accordance with Moore law, the number of CPU transistors, and therefore CPU performance doubles every 18 months), but the amount of random access operations hard disks can sustain per second increases only 10% a year.

Disk system performance for web-caching can be improved in many ways. First, and probably the most widely used way is to increase the number of disks. Web-caching software such as Squid is capable of evenly balancing read/write operations across disks. Another way is to use RAID arrays of disks, which usually have sophisticated controllers capable of distributing load and changing the sequence of read-write operations in order to improve average access speed. RAID arrays often have significant amount of RAM on-board which also helps to improve performance. For use with Squid, however, all these fancy RAID things are just a waste of money. Redundancy offered by many RAID types (e.g. RAID-1 and variations) is simply not needed for Web-caching application, because lost web-cache will recover itself with time (after disk replacement of course); write performance may be significantly degraded (especially true for RAID-5). The only suitable RAID array would be RAID-0 (striped disks), but it also does not provide any benefits over Squid's built-in disk load balancing (well, in case of RAID-0 there would be a single disk volume, but this will not improve Squid performance) and has the same level of reliability as no RAID at all. It is always better to spend money not to RAID controllers, but to buy additional disks and/or RAM. RAID-1 (mirrored disks) will add reliability at the cost of RAID controller and additional set of disks. However, when disk in RAID-1 fails, the computer containing it will have to be stopped to replace broken disk (no

information would be lost) and rebuild the array (in fact, formatting a new drive and creating a new Squid directory hierarchy on it usually takes *less* time than rebuilding RAID-1 array). During rebuild operation a significant part of the disk cache will most likely become stale. Therefore, there is no significant difference between RAID-1, RAID-0 and set of disks for web-caching purposes, except the cost. Yes, there are RAID-1 arrays with *hot swap* capability, which do not require turning computer off, but these arrays cost even more than ordinary RAID-1 arrays *and* after replacement of failed disk the performance of array is significantly degraded for a few hours during automatic array reconfiguration (information must be copied to the new disk).

Designing a highly-specialized filesystem is a way used in many proprietary solutions (NetApp, BlueCoat). General-purpose filesystems such as FFS in BSD, ext3fs in Linux or NTFS in Windows are not effective to be used in web-caches. Web-caching application usually requires disks to handle a significant amount of random access operations, but disks work best when reading data sequentially. Filesystems for web-caching operation can be improved in two ways. First, the amount of disk writes needed to store or replace an object, must be as low as possible. E.g. Novell Netware filesystem requires from 5 to 20 disk write operations to replace file on disk (Lee and Tomlinson, 1999). It is possible to optimize this operation, so it will always require just one write operation, and therefore write performance will be increased 5 times in worst case. Every cached web-page usually consists of many pages, so another way to improve disk access performance is to store all page's objects in one place at the disk if possible, because if one of page's objects is requested by user, it is likely that other objects will be requested as well, so the whole page could be read or written in one read/write operation, positively affecting performance. As shown by Lee and Tomlinson (1999), a page consisting of 10 objects can be written in 1 disk operation; comparing this to 50 write operations needed to accomplish the same in Novell NetWare filesystem, so we can conclude that it is possible to design specialized web-caching filesystem which would be 50

times more efficient than particular general purpose filesystem (which in turn is one of the fastest proprietary general purpose filesystems). Benchmarking specialized filesystems will require separate or significant changes to existing benchmarking tools such as Wisconsin proxy benchmark or Web Polygraph, because these benchmarks typically do not use real-life web-pages consisting of properly grouped objects (Lee and Tomlinson, 1999).

General-purpose operating systems are not optimized for Web-caching application. To achieve better results a special operating system is needed. NetCache Appliance (NetApp; Danzig, 1998) is a device built to be used as web-cache. It uses a special proprietary multithreaded event-based ONTAP kernel, which eliminates the need to copy data between filesystem and network stack[16], and is designed to handle many thousands of network connections efficiently. One of the most important features of Network Appliance is a special write-optimized filesystem – WAFL (Write-Anywhere-File-Layout). Conventional general-purpose filesystems such as FreeBSD's FFS or Linux ext2fs and ext3fs usually require several disk write operations to write a file to disk, delete or replace file on disk. In these filesystems not only file data must be written to disk, possibly requiring several write operations, but metadata such as file name, creation, modification and access times, ownership and permissions must be written or updated as well. Tweaking of filesystem parameters, such as mounting filesystem in asynchronous mode (unreliable, default setting for Linux ext2fs), using a journalled filesystem (in Linux) or FFS with *softupdates* (In FreeBSD – (FreeBSD; Ganger)) and disabling writing access times

[16] Most advanced open-source modern operating systems such as FreeBSD and Linux only recently got zero-copy support, which greatly reduces the number of context switches and improves overall network performance.

(*atime*) could significantly decrease the number of write disk operations required. But still general purpose filesystems will use more disk operations than WAFL or other highly specialized filesystem. Main design goal of WAFL was to keep number of disk operations as low as possible. WAFL filesystem is always consistent, so it does not require a long filesystem check (*fsck*) operations during reboot (well, ext3fs and FFS w/ softupdates do not require it either), it stores metadata with files. Due to the nature of web-caching a partially or incorrectly written data due to the system or hardware failure can be discarded on boot. All filesystem metadata are always kept in memory, so it is always possible to determine whether an object is present on disk without touching disk itself. Write operations in WAFL are highly efficient, requiring just one disk operation for a typical write or replace operation. According to Danzig's estimates WAFL is ten times more efficient for Web caching applications than traditional filesystems in terms of number of disk operations.

Central Processors (CPU) are not a bottleneck now, and are very unlikely to become such a bottleneck in the future. In recent few years CPUs have exceeded clock speeds of 3GHz and switched from 32-bit architectures to 64-bit (at affordable price), and their performance will be enough for a few years at least. In near future there would be new and powerful CPUs (probably multicore) available, and transition to 64-bit CPUs and OSes will soon be completed, so from the point of raw processing power we can feel safe. Current abilities of average CPUs significantly exceed the needs of most web-caching applications.

RAM is currently limited by 32-bit bus width of Intel IA-32 architecture (maximum of 4GB of addressable RAM), Operating Systems capabilities, number of available memory slots in typical computer (no more than 4) and RAM pricing (modules of 512MB DDR RAM have optimal price/performance ratio, everything with more density has prohibitive pricing at present time). Practical limit for commodity PCs at this time is around 2GB of RAM. In a couple of years a transition to 64-bit computers (AMD Athlon64 and Intel EMT64 architecture) will be completed, so

this limitation will cease to exist (or, at least will not depend on limitations of 32-bit architecture), so PCs with more memory will become practically feasible.

4.4 The Need for High-Performance Web-Caches

A single caching unit is preferable to multiple cooperating caches or caching hierarchy because the number of cooperating caches is limited by amount of inter-cache traffic, and storage capacity in web-caching hierarchies is not used efficiently. Even when deploying of web-cache hierarchy is justified by topology of network, it is desirable to have less nodes but powerful ones, than a big number of less powerful nodes. The higher the node is located in hierarchy, the more powerful it must be.

It is a known fact that Web-object popularity has Zipf-like[17] distribution (Breslau et al, 1999; Zipf, 1949; see also Fig. 7). One consequence of Zipf-like behaviour is that hit-rates have some upper limit, which is much less 100%. Documents belonging to popular part of distribution can be cached extremely effectively, but documents belonging to so called *heavy tail* of Zipf distribution will not be cached at all, because they will be accessed only a handful of times and will be superseded in cache by other documents according to LRU algorithm. Therefore increasing of web-cache storage capacity after some point loses any sense, because it leads to only logarithmic improvements in hit-rates.

---

[17] The probability p_i of request to i^{th} most popular document is proportional to $1/i^{\alpha}$, for some constant parameter α.

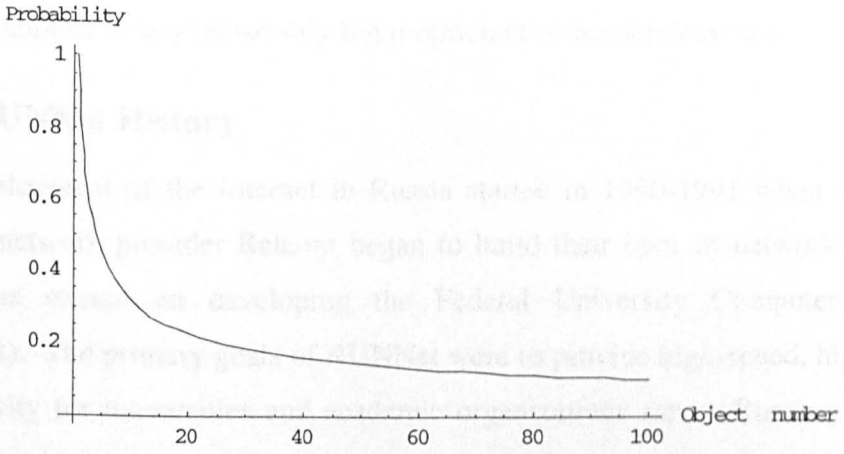


Figure 7. Zipf distribution.

Web-caches, which intercept requests to origin web-servers, cache most popular requests, changing traffic patterns observed by origin web-servers (Doyle et al), so upstream web-caches will contain most of the popular documents, so downstream caches will have to deal with less popular or uncacheable content. Doyle et al call it *Trickle-Down Effect.*

4.5 RUNNet as an Example of Typical Network

Russian University Network (RUNNet) is the biggest academic network in Russia. Due to its hierarchical highly centralized topology with a variety of different communication links (fibre optic, copper, satellite) and with thousands of users it is probably the ideal testbed for distributed web-caching hierarchy with a very high-performance cache at the top (natural location for this cache would be next to the most expensive communication channel). Author of this thesis has been working for RUNNet for quite some time as a system administrator maintaining and developing RUNNet web-caching system, so practical experience gained there has been valuable in web-caching research. RUNNet caching system is used in this thesis as a potential target for high-performance web-cache, so all performance requirements

are based on the needs of RUNNet caching system. These requirements however could be applied to any reasonably big proprietary or academic network.

4.5.1 RUNNet History

The development of the Internet in Russia started in 1990-1991 when the largest Russian network provider Relcom began to build their own IP network. In 1994 work was started on developing the Federal University Computer Network (RUNNet). The primary goals of RUNNet were to provide high-speed, high-quality connectivity for universities and academic organizations across Russia. From the very beginning the key difference between RUNNet and other Russian academic networks was the fact that RUNNet was relatively well funded. This allowed RUNNet to deploy its basic infrastructure in a few months. In March 1995 the RUNNet backbone was finished. Several federal nodes in Moscow, St. Petersburg, Yekaterinburg and several other cities were connected with satellite and land communication links. At that time, using satellite links was more cost-efficient, because of the vast geographic distances and poorly developed land communication links in Russia. For international links, connectivity was provided by Radio-MSU, one of the Moscow providers, to DFN, a German scientific network. Satellite communication link between Moscow and Hamburg has been established. This allowed access to American and European networks for all institutions connected to RUNNet. Peering agreements were signed with all major national Russian providers, thus, giving full access for RUNNet clients to all Russian and international Internet resources.

In 1996 in the second stage of RUNNet development several federal nodes were connected and a second international link, St. Petersburg-Lappeenranta (Finland) was put into operation.

In 1998 in the third stage Moscow and St.Petersburg were connected with a high-speed 155Mbps ATM link and the bandwidth of the link to Finland was increased to

4 Mbps. Towards the end of the year another international 6Mbps link, RUNNet-Teleglobe, was put into operation and the aggregate international bandwidth achieved 10Mbps.

At present time RUNNet has 622Mbps link to United States and Europe via FUNet/NORDUnet, and virtually unlimited bandwidth between Moscow and St.Petersburg. The number of universities and other education and research institutions grows at rapid pace. All these new developments require new level of performance, reliability and manageability from RUNNet web caching system.

At the time of writing RUNNet is undoubtedly the biggest and the most advanced academic network in Russian Federation, it provides Internet access to over hundred of research and educational institutions, so a proper design of network infrastructure and operating issues have a top priority for RUNNet.

4.5.2 RUNNet Architecture and Web-Caching System

At the present time, RUNNet is built using a two-star topology (see Figure 8) with the main nodes in St. Petersburg and Moscow. The St. Petersburg and Moscow nodes are connected to each other with several high-speed 622Mbps ATM communication links. The Moscow node gives RUNNet access to all Russian national networks through the M9 traffic exchange point, which is the crossing point for most Internet communications in Russia. The St. Petersburg node provides RUNNet with access to Internet providers in the north-western region of the Russian Federation and access to international networks. All international traffic currently goes through the St. Petersburg node. The RUNNet network operation centre is located in St. Petersburg Institute of Fine Mechanics and Optics. The nodes in Moscow and St. Petersburg form the RUNNet core network. RUNNet operates two of its own satellite ground stations located in St. Petersburg, which use satellites that cover most of the populated parts of Russia and several neighbouring countries.

RUNNet Federal nodes are connected to the core network via satellite links and where appropriately, via land communication links (usually fibre-optics). Typical connection speeds for satellite links vary from 64Kbps to 512Kbps with a maximum possible speed of up to 4Mbps. Land communication links have a bandwidth from 512 to 2048Kbps and more. Federal nodes located in most big cities across Russia including Barnaul, Vladivostok, Yekaterinburg, Izhevsk, Irkutsk, Krasnoyarsk, Makhachkala, Nalchik, Nizhny Novgorod, Novosibirsk, Pereslavl-Zalessky, Perm, Petrozavodsk, Rostov-on-Don, Saratov, Stavropol, Tambov, Tomsk, Ulyanovsk and Khabarovsk form the backbone of the RUNNet network. There are many institutions connected to the federal nodes through regional IP networks. Federal nodes serve as centres of regional academic network development giving them access to Russian and international networks. In recent years, federal nodes and the regional networks connected to them grew up very intensively. Federal nodes are usually equipped with a satellite transceiver station, backbone router, node server and regional router. The backbone router connects to one of two core nodes. The regional router provides access to the regional network. This architecture has been proven as highly flexible and effective during the last several years of RUNNet operation.

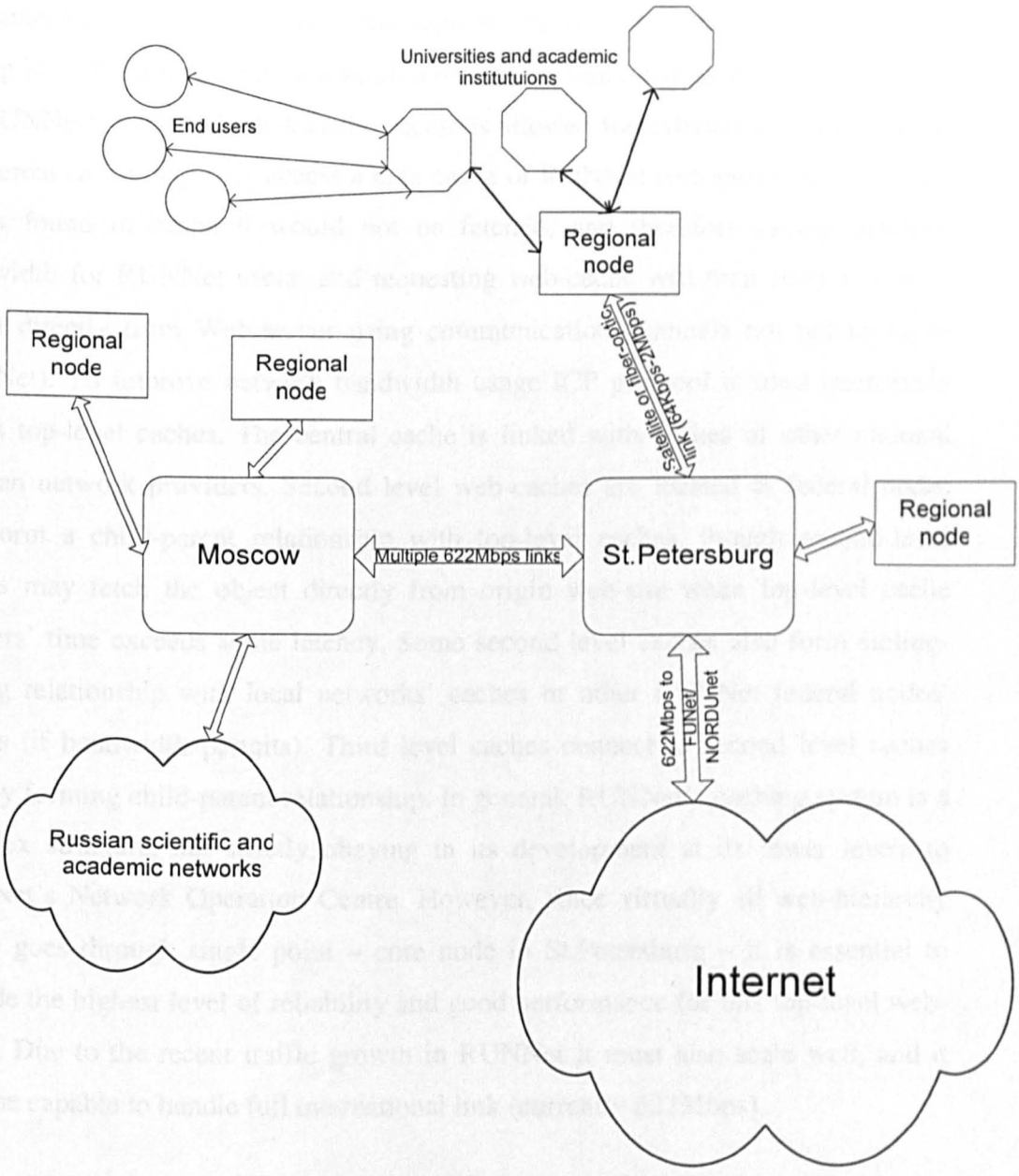


Figure 8. RUNNet architecture.

The overall design of the RUNNet caching system resembles the design of RUNNet itself (see RUNNet web-site for further information). The RUNNet caching system forms a 3-level hierarchy and its top level consists of a central web cache located in St. Petersburg and second-level caches in federal nodes. Third level caches can also

be connected to federal nodes in the regions. First-level caches[18] form a sibling-sibling relationship, sometimes a limited one (for external institutions not connected via RUNNet), when only disk-cache access is allowed for external web-caches (that is external cache may only access a disk cache of RUNNet web-proxy, and if object is not found in cache it would not be fetched, and therefore saving precious bandwidth for RUNNet users; and requesting web-cache will then receive a web-object directly from Web-server using communication channels not belonging to RUNNet). To improve network bandwidth usage ICP protocol is used intensively across top-level caches. The central cache is linked with caches of other national Russian network providers. Second level web-caches are located at federal nodes and form a child-parent relationship with top-level caches, though second-level caches may fetch the object directly from origin web-site when top-level cache answers' time exceeds some latency. Some second level caches also form sibling-sibling relationship with local networks' caches or other RUNNet federal nodes' caches (if bandwidth permits). Third level caches connect to second level caches usually forming child-parent relationship. In general, RUNNet's caching system is a complex structure, not strictly obeying in its development at its lower levels to RUNNet's Network Operation Centre. However, since virtually all web-hierarchy traffic goes through single point – core node in St.Petersburg – it is essential to provide the highest level of reliability and good performance for this top-level web-cache. Due to the recent traffic growth in RUNNet it must also scale well, and it must be capable to handle full international link (currently 622Mbps).

[18] At present time there is just one operational top-level web-cache in St.Petersburg

Most web-cache hierarchy nodes use different kinds of Unix and Linux OS on PC or Sun Microsystems platform, and Squid as web-caching software, and therefore compatibility problems do not arise. Some third-level caches may use other solutions at their choice; some use Windows- or Novell Netware-based proprietary software, which might be more convenient for them. This software usually interoperates flawlessly with squid-based web-cache nodes both via HTTP and ICP.

4.6 Analysis of Statistical Data of Web-Caching System in RUNNet

There is always not enough bandwidth. RUNNet is no exception to this rule. The bandwidth of domestic links is however much easier and cheaper to increase, but international links is a different matter. The cost of international links is sometimes an order of magnitude higher comparing with domestic traffic. Thus it is crucial to achieve a high utilization and high efficiency of international links. Web-caching is one of the solutions which allows to implement the same level of service at lower price. To study the ways to improve web-caching system in RUNNet an internal unpublished research was performed, which has shown that:

1. The traffic is likely to grow at rapid pace, and international traffic growth is seriously limited by available international bandwidth (see Figure 9. 4Mbps RUNNet/FUNet international link for a two-year period.); domestic traffic grows exponentially (see Figure 10. Moscow-St.Petersburg RUNNet channel statistics (day averages).)
2. Most of all traffic is HTTP, FTP usage declines every year, so it is most important to effectively deal with HTTP, and all other protocols can be ignored by web-caching system.

The existing web-caching system does not perform well because most of the cacheable traffic passes web-caching system due to the inability to control user's browsers. The exponential growth of international traffic in RUNNet was always limited by the maximum bandwidth of international link. The Figure 9 shows that at

the time of study the international link (4Mbps at that time) was heavily overloaded, so it was not possible to predict usage patterns for the following years. In addition, web-cache behaviour in low bandwidth conditions differs significantly from normal non-overloaded situation (Hada et al, 1999). However at the same time the traffic at RUNNet backbone link between Moscow and St.Petersburg was not limited by the link capacity, and this traffic grew exponentially (Figure 10).

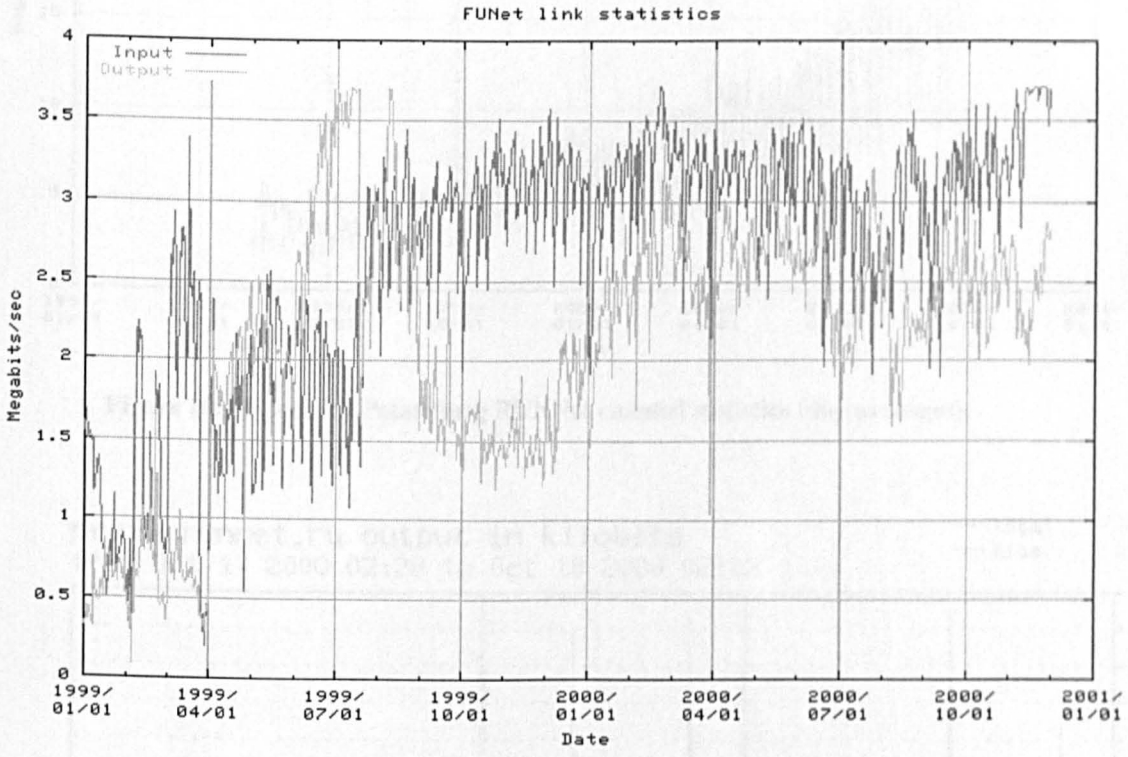


Figure 9. 4Mbps RUNNet/FUNet international link for a two-year period.

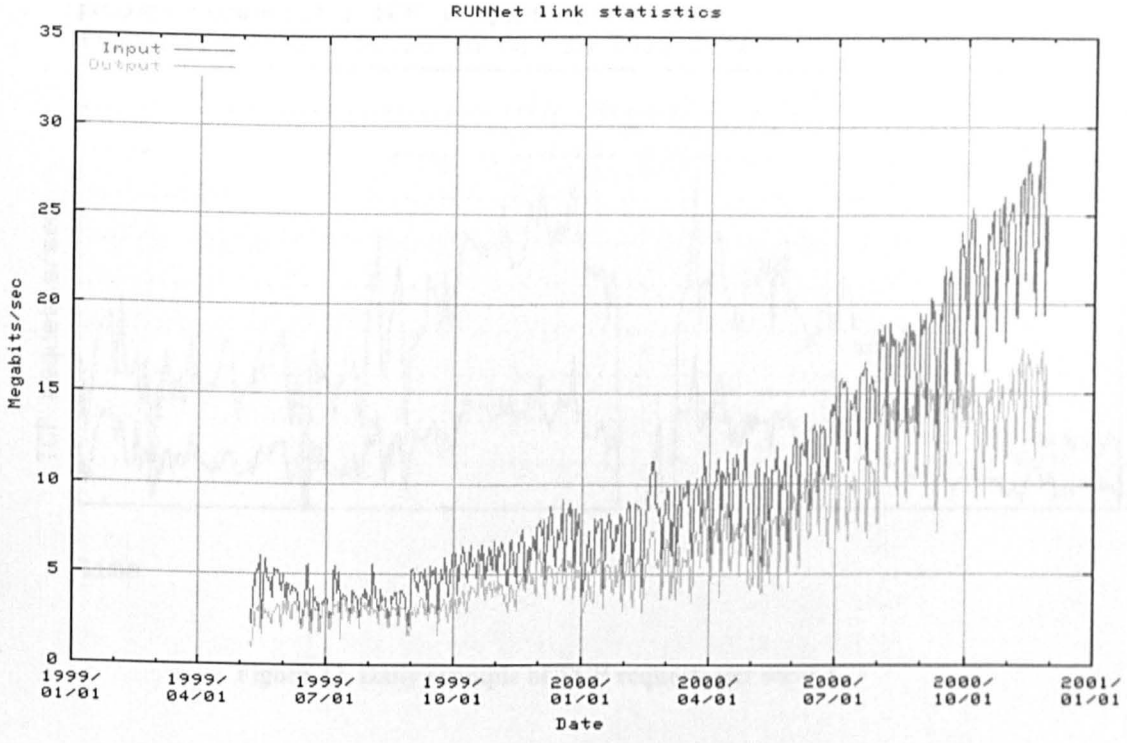


Figure 10. Moscow-St.Petersburg RUNNet channel statistics (day averages).

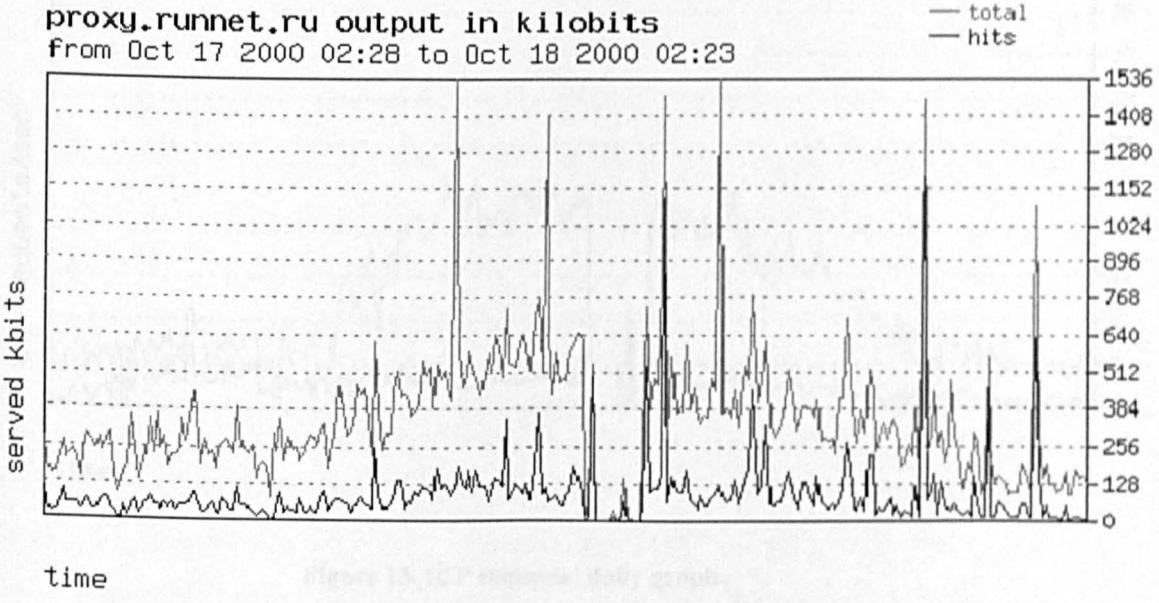


Figure 11. Primary RUNNet cache daily output example.

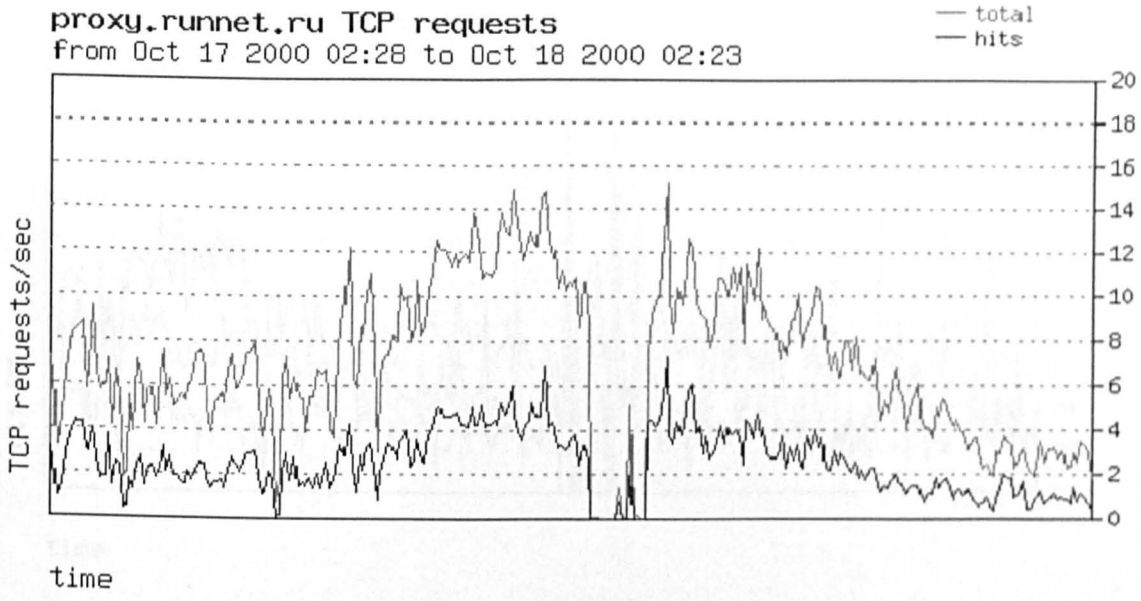


Figure 12. Daily example of TCP requests per second.

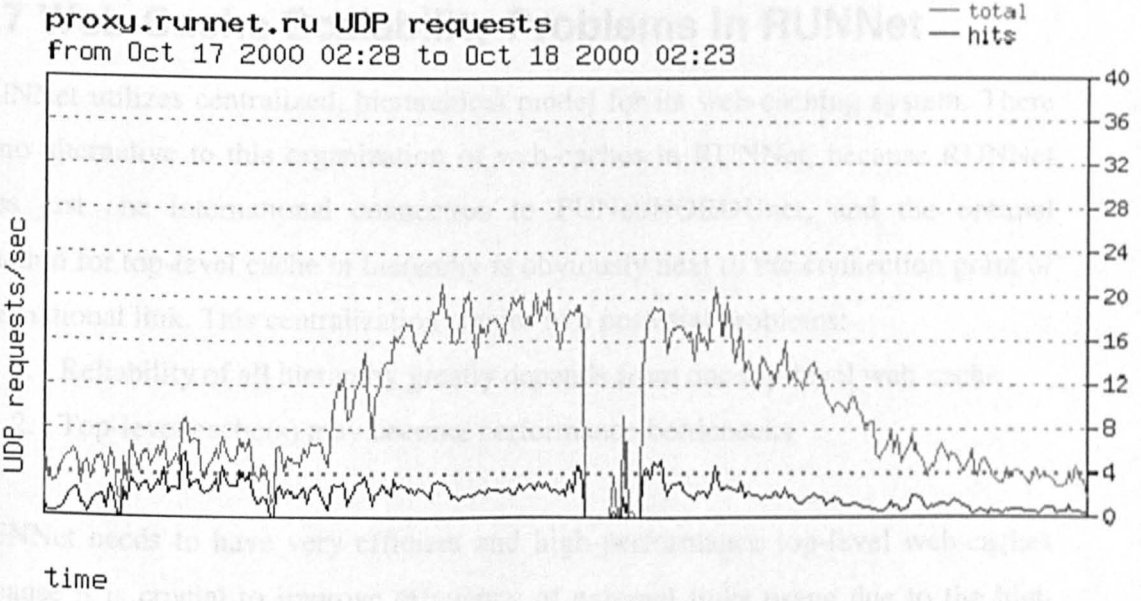


Figure 13. ICP requests' daily graph.

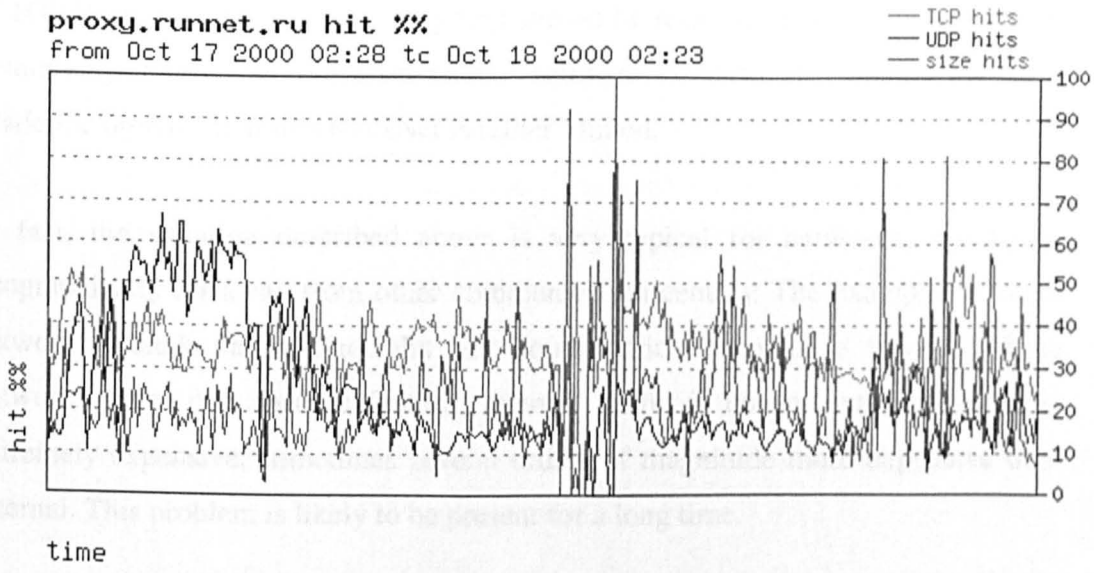


Figure 14. Hit rates (for HTTP and ICP) and size hits for HTTP.

4.7 Web-Cache Scalability Problems in RUNNet

RUNNet utilizes centralized, hierarchical model for its web-caching system. There is no alternative to this organization of web-caches in RUNNet, because RUNNet uses just one international connection to FUNet/NORDUnet, and the optimal location for top-level cache in hierarchy is obviously next to the connection point of international link. This centralization creates two potential problems:

1. Reliability of all hierarchy greatly depends from one top-level web-cache
2. Top-level cache(s) may become performance bottlenecks

RUNNet needs to have very efficient and high-performance top-level web-caches because it is crucial to improve efficiency of external links usage due to the high cost of those links. At the same time the task of building hierarchy becomes second-priority task, which is well studied and could be solved by using proven methods. So the main task is clear: build a high-performance, reliable, extendable, manageable web-caching system capable of handling several hundreds of megabits

of HTTP traffic. The cost of this system should be reasonable; it should use cheap commodity hardware and open-source software because the budget of such academic organization like RUNNet is rather limited.

In fact, the situation described above is very typical for networks, which are geographically far away from other communication centres. The examples of such networks include Japan, Australia and South Africa, to name a few. All these networks share one specific feature: internal traffic is cheap, external traffic is extremely expensive, sometimes several orders of magnitude more expensive than internal. This problem is likely to be present for a long time.

4.8 RUNNet Web-Caching System Development

Current state of web-caching system in RUNNet is pretty well the same as few years ago. There is a single web-cache with four SCSI 10000RPM 18GB disks and 1GB of RAM, which handles up to 50GB of data, up to 10 million requests per day (a typical daily traffic graph can be seen at Figure 11. Primary RUNNet cache daily output example.; HTTP request graph at Figure 12. Daily example of TCP requests per second.; and ICP request graph at Figure 13. ICP requests' daily graph.). RUNNet proxy achieves up to 35% daily average object hit rate, and up to 20% size hit rate (see Figure 14. Hit rates (for HTTP and ICP) and size hits for HTTP. for a typical daily graph of hit rates). A few second level caches are linked to this central cache, mostly caches from universities connected via satellite links. 50GB per day is roughly equivalent to 5Mbps stream, what means that most of the data travelling through international link does not even have a chance to be cached. After recent upgrade of international link to 622Mbps at least 500Mbps of traffic could be cached (recent unpublished study in RUNNet has shown that web traffic constitutes up to 80% of total traffic). It is obvious that present web-caching system is terribly ineffective. Not so many organizations connected to RUNNet deploy their own second-level web-proxies due to various reasons, and the end-users only in rare cases know about existence of web-cache.

The only way to direct most of web-traffic through web-cache is to implement some sort of transparent web-caching. This will make most of the web traffic to go through central web-cache, which will have to handle up to 500Mbps of international traffic plus at least the same amount of domestic traffic. This will give us a requirement to handle a sustained rate of up to 250Mbps (day average is around 125Mbps) at present time and up to 500Mbps (day average ~ 250Mbps) in two-year time. Handling 500Mbps of traffic is a very demanding task. The best proprietary solutions to web caching (NetApp, Cisco, BlueCoat) are capable of handling only 100Mbps per hardware unit. To achieve high hit rates it is considered a good practice to cache 7-day amount of traffic, however we will accept that it is enough to store just 1 day of traffic. For 250Mbps day-average stream it will give us up to 2TB of storage (counting weekends as just one day instead of two, because traffic in weekends is significantly lower than in weekdays).

So we have the traffic and storage requirements to the ideal future web-caching system of RUNNet:

1. Ability to handle day-average stream of 250Mbps with peak load of 500Mbps.
2. Storage of 2TB.

These requirements might seem a little bit exaggerated, but they are in fact upper estimates, which most likely will never be achieved in real life (provided the link capacity stays the same of course). However they could constitute a good task to accomplish. It is believed that there are quite a few companies and academic institutions, which have the same or higher requirements to web-caching system, so the resulting solution might be helpful for them.

Due to the various reasons RUNNet cannot use expensive, closed, proprietary solutions provided by companies like Netapp, Cisco and the like. These reasons

include high price, limited and expensive support (at least in Russia), high cost of upgrades, to name a few. The choice of hardware and software is rather limited. Commodity x86 architecture cheap hardware should be used, because for most applications it provides better price/performance ratio (though with lower reliability/durability), it is also much easier to obtain and to support in comparison with non-Intel compatible architectures. For example, in Russia the cost of AMD Athlon XP 2500+ system with 1GB of memory, one 1000BaseT and one 100BaseT Ethernet interface, and two 160GB IDE ATA-133 hard drives is well below 1000USD. The choice of OS is also clear – there should not be any licensing fees, the software should be cheap. Open-source free operating systems such as FreeBSD and Linux are ideal candidates.

The resulting web-caching system should scale well, that is almost linearly; it also should allow smooth increase in performance when it is needed. Maintenance and operating expenses, sometimes referred to as TCO (total cost of ownership), should also be acceptable. And the last but not least requirement is high reliability, the future web-caching system must quickly recover from most types of failures in fully automatic mode, without human intervention, because it is not always possible (e.g. due to the budget limitations) to have 24x7 support on-site. Taking into account that web-caching system is to be operated in transparent web-proxy mode, the reliability is a crucial quality for web-caching system to be successful.

As shown in the following chapters the proposed design for web-caching system ideally suits these requirements. Provided there is a will to implement high-grade web-caching system in RUNNet, and there are enough funds, the proposed solution will greatly improve the efficiency of international and domestic links usage, so the net effect of web-caching system deployment would be significant monetary savings.

4.9 Summary

To achieve higher hit-rates it is important to direct as much of HTTP traffic to Web-cache as possible. In any organization with more than ten people this task can be achieved only by using transparent web-caching. Building web-cache, suitable to handle even modest load, can be complicated by technical difficulties such as limitations of hard drives, RAM and CPU, as well as by reliability and cost requirements. In general, to handle every 8Mbps of HTTP traffic a single 40GB disk, 512MB of RAM and 500MHz CPU is required. Web-cache with performance up to several hundreds megabits would require numerous disks, multiple CPUs and significant amount of RAM. Such high-performance web-cache would be needed in any medium to large network, and Russian University Network (RUNNet) is an ideal place for high-performance web-cache. This cache will have to meet highest performance, reliability, scalability, extendibility requirements; it will have to have the best possible price/performance ratio.

CHAPTER 5

BUILDING A HIGH-PERFORMANCE WEB-CACHING CLUSTER

Web-caching is a special application of cluster technology, which has some unique features, that must be taken into account when designing web-caching clusters. Universal or general solutions may not work efficiently when applied to clusters of web-caches if not adjusted accordingly. First, web-cache content is a special kind of data. It is valuable, because a significant amount of bandwidth has been used to get it, but at the same time it can be lost at any time without any dramatic consequences (though it is not desirable, because it will negatively affect cache hit ratio, increase user perceived latencies and bandwidth consumption) and later recovered. Second, the consistency of cached objects across several web-caches taking part in hierarchy or inside a cluster is also not terribly important due to the nature of web-objects, which sometimes do not have a predefined expiration time, so web-caches sometimes make assumptions about it. Third, the Web traffic itself is bursty in nature[19], so clustering software must provide mechanisms to dynamically distribute load among the nodes (Baker and Moon, 1999). Fourth, most commonly used scheme for deployment of web-caching clusters is transparent caching, which makes

[19] Some sites may suffer from so-called 'Slashdot effect', which is called after popular Slashdot.org site, which publishes links to external resources. Since Slashdot.org has a lot of readers, these people can bring almost any web-server to its knees by clicking on provided links to that web-server.

impossible direct access to the Web resources, because all requests to remote Web-sites are redirected to transparent cache by a router. Therefore reliability (and failover capabilities) of web-caching cluster have a significant importance. All these factors should be taken into account when dealing with web-caches or hierarchies to achieve better results.

5.1 Cluster Approach to Web-Caching

The idea of using clusters to build high-performance web- and streaming caches is not new (Chen and Zhang, 2002; Cherkasova and Karlsson; Chung and Kim, 2001; Menasce, 2002).

Guo et al (2003) propose a reconfiguration strategies for a cluster of caches for video streaming. They use a cluster topology with a selected node which interacts with users, computes optimal object placement, balances load across nodes by moving some objects from one node to another.

Feenan et al (2002) describe an approach to build a reverse proxy cluster used in Oracle9iAS Web Cache. In proposed scheme all nodes are symmetric, that is they have the same hardware configuration and are expected to handle the same load. Each node in cluster handles its own part of web object space; partitioning is based on analyzing URL of web-objects and optionally some metadata such as HTTP headers or session parameters.

5.2 Cluster Communication Protocols

There exist a number of cluster state synchronisation protocols, some of them are used in practical applications, and some of them are strictly theoretical as yet (Borcea et al, 2002; Casalicchio and Colajanni, 2001). Kondou et al (2002) propose

a self-stabilizing PIF[20] protocol with fault recovery for tree-like networks, which stabilizes in *O(h)* time[21]; after stabilization, information propagation time takes *O(h)* rounds, and recovery from single failure takes *O(1)* rounds. Cournier et al (2002) present a PIF protocol suitable to be used in arbitrary networks. In this protocol any node of the network can initiate synchronization phase by sending a broadcast message or *wave* to its neighbours - *children*, which in turn send a message to their neighbours with exception of node – *parent* - from which original message was received. When node receives a confirmation of delivery from its children, it sends a confirmation to its parent. Cournier et al. propose a snap-stabilizing version of this algorithm, which always behaves in accordance with its specification.

Ethernet medium does not provide reliable transmission time for data messages (Kweon and Shin, 2003). Some messages may be lost; other messages may be delivered much later than average transmission time. The reason is the very nature of Ethernet, the principle it is based upon – CSMA/CD (Carrier Sense Multiple Access/Collision Detection) protocol. Collisions in Ethernet may be eliminated by using full-duplex mode (virtually every 100Mbps Ethernet network card available today is capable of full-duplex mode; original Ethernet uses simplex mode, that is in a pair of hosts at any given moment of time only one host can transmit data and another host can only receive data) and deploying Ethernet switches. The Ethernet switch is more complex and expensive than Ethernet hub, however this price difference (for 100Mbps Ethernet) is so minor, that we can assume using full-duplex Ethernet switch for cluster state interconnection. As it shown by Kweon and Shin (2003) even traditional simplex 100Mbps Ethernet is perfectly suitable for cluster

[20] Propagation of Information with Feedback.

[21] *h* is the height of the tree network.

state distribution purposes under the condition of no more than 2 full-size Ethernet frames per second per node.

Cluster communication protocols must provide reliable ways of message delivery (Amer et al, 2002; Andrews et al, 2002). Usually this goal is achieved in group communication protocol itself, but not in underlying protocol, because the most commonly used underlying protocols such as UDP, IP or Ethernet do not give any delivery guarantees. TCP protocol does provide reliable delivery, but it is quite heavy-weight; when some packet is lost TCP would try to retransmit that packet a few times; this process may take up to several seconds until error will be indicated. Most cluster protocols do not require such advanced features of TCP as rate-limiting etc., so often much simpler UDP is used as a basis for cluster communication. Some messages may be lost due to the physical media error or Ethernet switch queue overflow, or may be discarded due to the high load of one of the hosts. Group message protocols must provide recovery from these situations, which is usually accomplished by error detection and message retransmission. However, there is a possibility of a permanent network failure, so cluster nodes would lose connectivity with each other for an extended period of time. Reaction to this event may depend on the nature of application; in some cases nodes may continue to function, but in other cases they must be shut down. To guarantee recovery from permanent network failures network connection must be redundant, which is relatively cheap for local area networks. There exist a number of group communication protocols such as Totem Redundant Ring Protocol (Koch et al, 2002). Totem RRP is similar to Token Ring LAN protocol, but it is focused on providing node and network failure detection using multiple Ethernet network interfaces. Totem RRP uses token passing, so at any moment of time only one node holding a token can transmit data, and all other nodes are listening at that time. Token Ring-like protocols based on Ethernet allow very fast token circulation, because collisions do not happen, so Ethernet can operate at speed close to its maximum theoretical value. E.g. Totem RRP is capable of sustained rate of 9000 1KB messages per second circulation in shared Fast

Ethernet network (100Mbps), which allow for a big number of participating nodes and/or very fast failure recovery times.

5.3 Load Balancing

Scalable web-caching can be done in many ways (Bourke; Kopparapu). The simplest and probably the oldest one is to deploy multiple separate or connected web-caches and use proxy-autoconfiguration in web browsers (Netscape) and use a simple hash-function[22] on URL to distribute load between separate caches (Danzig, 1998). This approach allows to redistribute load between separate caches (Aversa and Bestavros, 2000), but it requires manual or automated configuration of every user's browser, and every user might override this setting, and therefore bypass cache; it also slightly increases page download times, because for every object download a JavaScript function has to be executed. In general, administrative issues render this approach completely unusable in modern conditions. More practical approach is load balancing at the router or Layer-4 switch[23].

Load balancing in clustered web caches can be done in several ways.

1. Random Load Distribution (RLD). Load information of the nodes is not taken into account; node is either selected using some sort of PRNG[24] or using a hash function applied to some parameters of the request, so every node has an equal chance[25] to serve a request. This assumes that the nodes

[22] This hash function might be simple, e.g. the sum of all URL symbols modulo number of web-caches.

[23] Operates at layer 4 of OSI model (transport layer).

[24] Pseudo Random Number Generator.

[25] Provided PRNG and hash function are good enough.

should have identical configuration. RLD works surprisingly well in many situations.

2. Round Robin Distribution (RRD). Load information of the nodes is not taken into account; request routing decision is made basing on previous routing decisions. E.g. nodes can get requests sequentially from first to last, and so on. Round Robin Distribution is supported and usually implemented with DNS software, and does not really require any cluster software. When using DNS RRD, every node of the cluster has a different IP address, and all these addresses correspond to the same domain name. The problem with DNS-based Round Robin Distribution is the fact, that when one of the nodes goes down or is taken offline, DNS configuration must be updated, and this update might propagate to users hours later due to the DNS caching, so for some of the clients service might be unavailable for a long time.
3. Load-Based Distribution (LBD). Request routing decision is based on load of the cluster nodes. A typical Load-Based Distribution implementation will choose a most lightly loaded node to serve incoming request. LBD is more complex than two previous distributions, because a state of the nodes must be maintained and synchronized.

Bunt et al (1999) study effects of choosing a particular load balancing technique for the purposes of web-caching cluster using trace-driven simulation. They conclude that simple load distribution policies such as Random Load Distribution or Round-Robin Distribution may work well provided every request's bandwidth is somehow limited (which is always true in practice, because web-caches have a numerous relatively low-bandwidth clients and not few high-bandwidth ones). Another consequence of this simulation is that Load-Based Distribution should be used to build a cluster with a big number of nodes, for small number of nodes Load-Based Distribution does not provide any significant benefits over Random Load Distribution, while being much more complex.

5.4 Recovery from Failures

When one of the nodes of the cluster fails, loses connectivity or is taken offline, some connections to it may be terminated, causing disruption to users' service. This is especially undesirable for critical applications like financial database transactions or other important data transfer (Aghdaie and Tamir, 2001). For some applications it is possible to redirect user requests to another server (Cardellini et al, 2003). For streaming content termination of connection will force user to reconnect, which is often a long and painful process. Sultan et al (2002) proposed a method and implemented it in software prototype to migrate established TCP connections from one server to another. In many cases though this connection migration is not worth effort needed to implement it. Changes to server OS and application software are required, which increases complexity of the whole system and potentially makes it less reliable and more difficult to maintain and support. Web-caching is an area where connection migration is not needed, because web-clients are expecting data connection termination at any time and can handle this situation properly. From user's point of view, in worst case the user will have to hit Reload button.

5.5 Summary

Single computer capable of handling a few hundred megabits of HTTP traffic is possible, but not cost effective; a group of standalone computers communicating with each other using ICP or HTCP protocols will not provide required level of reliability, load balancing. Specialized web-caching cluster is the only way of building high-performance Web-cache in a cost efficient way. Web-caching cluster requires a special kind of cluster protocol, which takes into account unique characteristics of web-caching application. The main difference between web-caching application and most other applications (database, web serving etc.) is that the contents of the web-cache can be lost without any grave consequences (though it is not desirable). Load balancing can be achieved using several approaches, including Random Load Distribution, Round Robin Distribution and Load-Based Distribution. The latter is most suitable for a cluster with big number of nodes. Web-

caching cluster must provide automatic fault detection and recovery mechanisms; failure of any node in the cluster will not be visible by web-cache user.

CHAPTER 6

PARTICULAR IMPLEMENTATION OF CLUSTER APPROACH FOR WEB-CACHING

6.1 Design Goals

One of the main objectives of this research was to provide a *realistic* way to build high-performance web-caches providing the best possible price/performance ratio. This requirement greatly reduces possible choices both in terms of hardware and software. First, proposed solution must use cheap hardware, widely available virtually everywhere. Second, it must use as much as possible existing popular software, so network personnel, who will deploy and maintain this web-caching software, would benefit from their existing network and system administration skills; and that software can not be expensive. Third, proposed solution must require minimal installation and maintenance efforts. Fourth, resulting web-cache must provide high performance, scalability, reliability, low price/performance ratio. As shown below, proposed web-caching architecture satisfies all these requirements.

6.2 Performance Requirements

To handle 500Mbps of HTTP traffic for extended periods of time resulting cache must conform to the following requirements:

1. 64 IDE disks with 40GB allocated to disk cache each (totalling 2.5TB).
2. Single 2GHz CPU for every 2 or 4 disks (32 or 16 total).

3. 512MB of RAM for every disk (32GB total).

A single computer conforming to above mentioned requirements is real, but not cost-effective. A cluster of cheap commodity PCs will do the job at a fraction of cost of a single computer. The possible cluster's configurations would be 16 identical nodes each having 2GHz CPU, 2GB of RAM, 4 80GB IDE drives with 40GB per drive allocated to disk cache (remaining space would be used by OS, web-cache logs, swap files etc.) plus two network interfaces (one Gigabit and one Fast Ethernet). Node configuration with two disks each and half the amount of RAM is less cost effective (it will require 32 nodes). The cost of each node in proposed configuration can be lower than $1000 and the total cluster cost (nodes themselves plus relatively expensive networking hardware such as Gigabit switch/hub; Fast Ethernet switches are cheap these days) can be less than $20,000. In comparison, the cost of Sun Fire 4800 Server in identical configuration would be an order of magnitude higher. A single server will also require special web-caching software benefiting from multiple CPUs, Squid will not fit there.

6.3 Implementation Specifics

Web caching cluster is ideal architecture for high-performance web-cache. Clustering approach is used with great success in many applications, including web-servers, databases, high-performance applications and web-caching software as well, which is easily explained by advantages of cluster technology. First, clusters are usually built using relatively cheap nodes, so overall cluster performance may be increased trivially and usually almost linearly, and cluster overall building costs are usually low in comparison to single device capable of running at the same performance level as cluster. Even though nodes' reliability may be low, overall cluster's reliability and fault-tolerance when cluster is implemented properly may be very high due to the cluster control software, which can monitor nodes, distribute load among them, and switch off failed nodes.

6.3.1 Cluster Structure

For web-caching purposes a symmetric cluster is proposed, where all nodes have the same role and rights (Fig. 15). This cluster requires several nodes, which are connected via network (via external network interfaces of cluster's nodes) to the normally configured router; separate network interconnection (via internal network interfaces of cluster nodes) between nodes is also required.

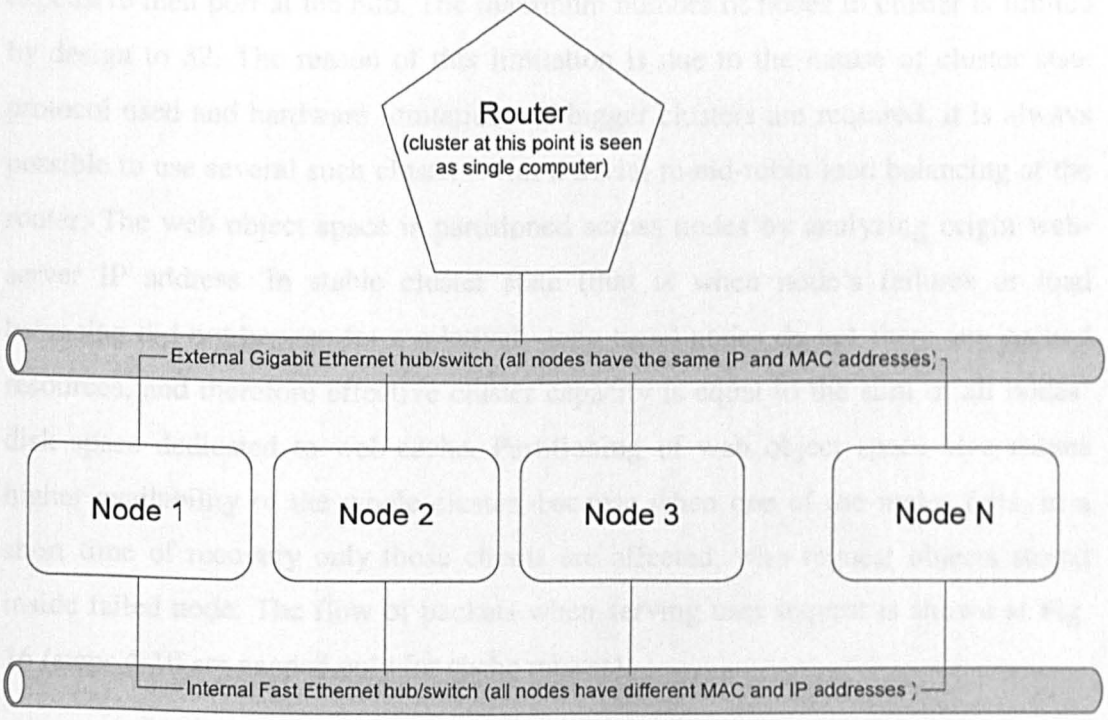


Figure 15. Cluster topology.

Main feature of the proposed cluster is that it uses *the same* IP and MAC addresses on external interface, so this cluster looks like single host even at the nearest router; every cluster's node receives all amount of traffic destined for the whole cluster, and then using a special technique it selects requests intended for this particular node and handles these requests, returning reply to the client via the same external interface. Internal network interfaces have different IP and MAC addresses and are used for cluster state protocol and for system administration purposes. This cluster

configuration does not require any unusual configuration of any network equipment (routers and hubs) with a possible exception of Ethernet switch used for nodes' external interfaces – this switch must have capability to forward Ethernet frames to multiple ports (Gigabit Ethernet hub would not require any special configuration). It is also possible to connect nodes directly to router via gigabit interfaces, but that would be too expensive, because ports at the router are usually much more expensive then port at the hub. The maximum number of nodes in cluster is limited by design to 32. The reason of this limitation is due to the nature of cluster state protocol used and hardware limitations. If bigger clusters are required, it is always possible to use several such clusters with a trivial round-robin load balancing at the router. The web object space is partitioned across nodes by analyzing origin web-server IP address. In stable cluster state (that is when node's failures or load balancing did not happen for a relatively long time) nodes do not share any cached resources, and therefore effective cluster capacity is equal to the sum of all nodes' disk space dedicated to web-cache. Partitioning of web object space also means higher availability of the whole cluster, because when one of the nodes fails, in a short time of recovery only those clients are affected, who request objects stored inside failed node. The flow of packets when serving user request is shown at Fig. 16 (steps 4-10 are needed only for cache misses).

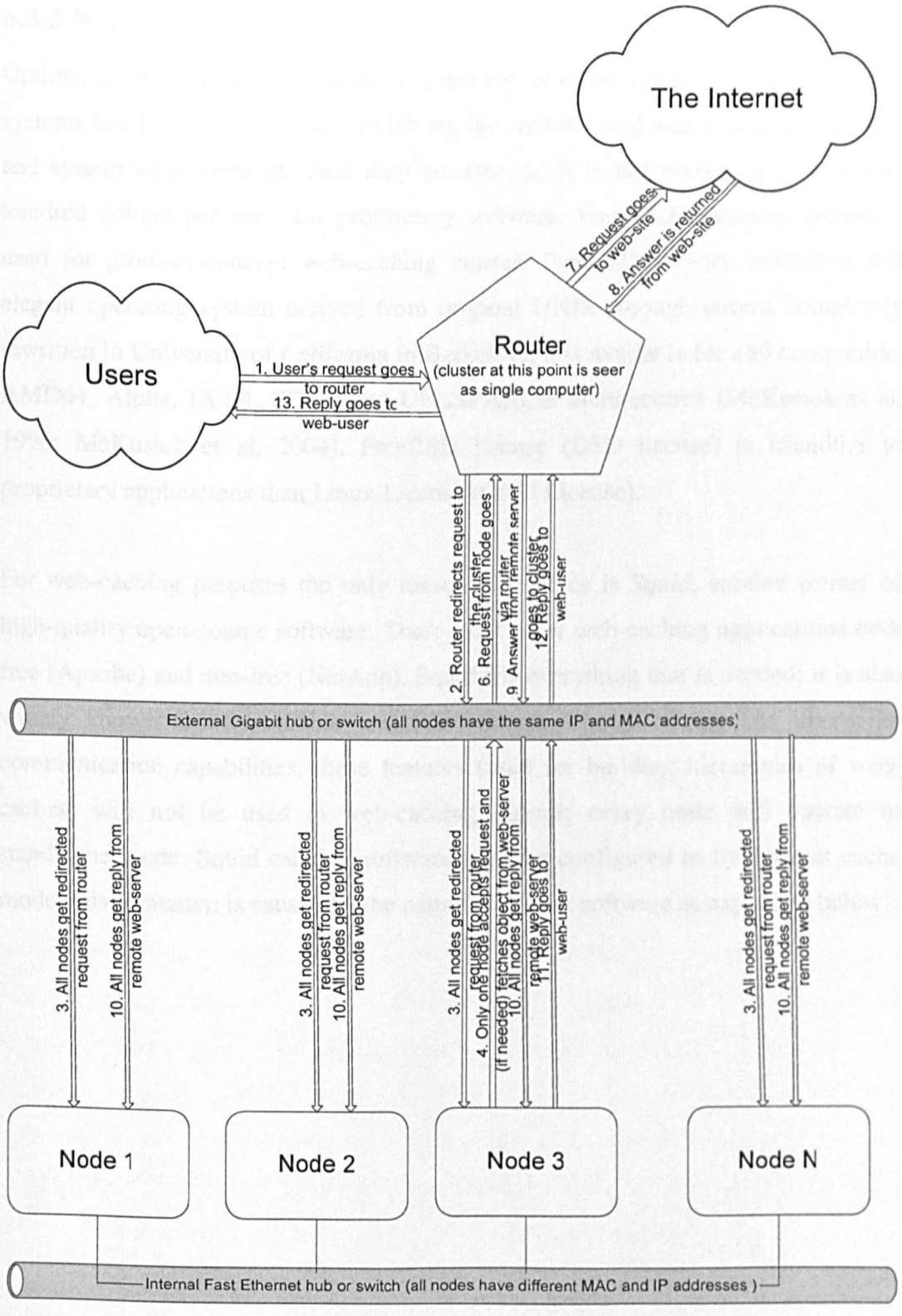
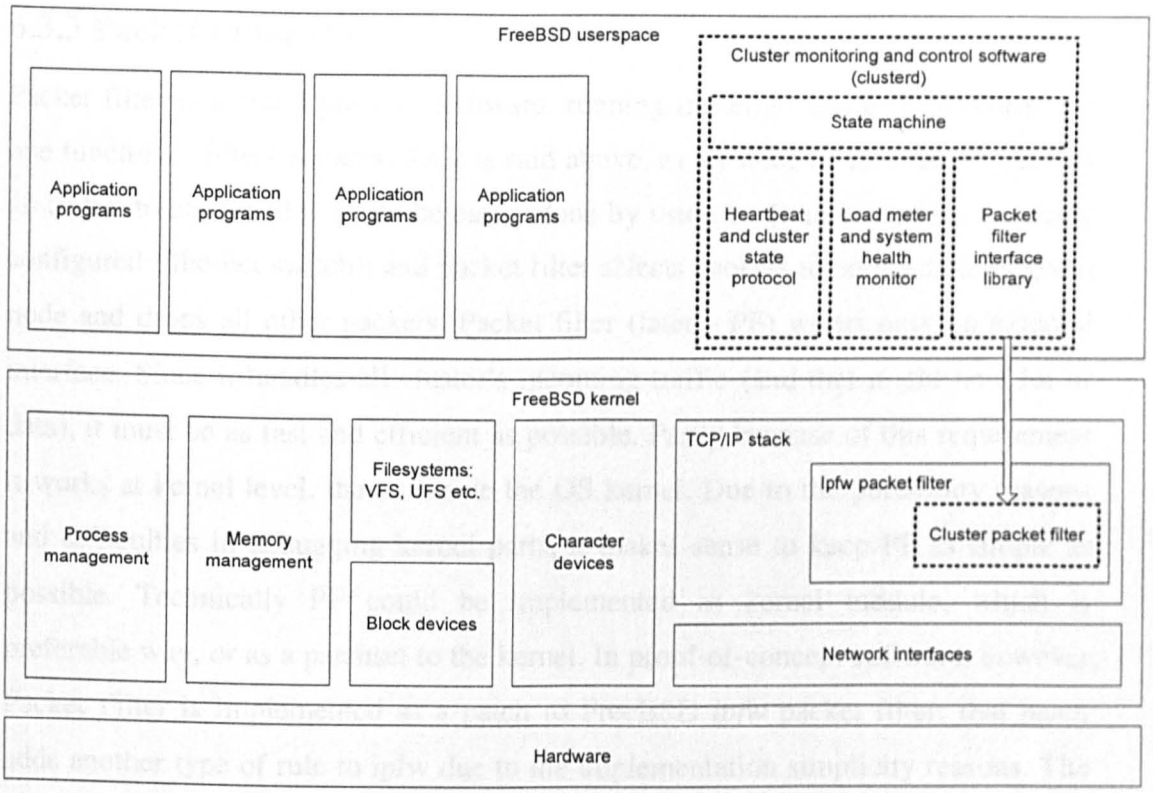


Figure 16. Request flow.

6.3.2 Node Software

Optimal software for cluster's nodes is some sort of open-source Unix-like operating systems like FreeBSD or Linux, which are fast, reliable and well known by network and system administrators. And they are free, so it is not needed to pay several hundred dollars per node for proprietary software. FreeBSD operating system is used for proof-of-concept web-caching cluster. FreeBSD is very consistent and elegant operating system derived from original UNIX (though almost completely rewritten in University of California in Berkeley); it is available for x86 compatible, AMD64, Alpha, IA-64, PC-98 and UltraSPARC® architectures (McKusick et al, 1996; McKusick et al, 2004). FreeBSD license (BSD license) is friendlier to proprietary applications than Linux License (GNU License).

For web-caching purposes the only reasonable choice is *Squid*, another primer of high-quality open-source software. There exist other web-caching applications both free (Apache) and non-free (NetApp). Squid has everything that is needed; it is also widely known and is reliable and fast enough. Though Squid has intercache communication capabilities, these features (used for building hierarchies of web-caches) will not be used in web-caching cluster; every node will operate in standalone mode. Squid caching software must be configured in transparent cache mode; this limitation is caused by the nature of cluster software as explained below.



Cluster software parts are shown in dotted boxes.

Figure 17. Cluster software components.

Cluster control software, which distributes load, provides fault-tolerant capabilities, is a heart of high-performance cache. It consists of 2 main parts – low level packet filter and high-level control software (see Fig. 17). High-level control software runs as daemon in user mode, it keeps cluster state synchronized among all nodes, controls packet filter, monitors node's health and load, reacts to miscellaneous events such as excessive load to one of the nodes, introduction of new node to the cluster, lost contact with one of the nodes, software failure or some sort of hardware problems at one of the nodes, manual switching of one of the nodes offline or for maintenance. Every node at any given point knows state of every other node in the cluster. This state is tightly kept synchronized among all nodes.

6.3.3 Packet Filter (PF)

Packet filter is a small piece of software, running in kernel mode. It provides just one function – filters packets. As it is said above, every node of the cluster receives all traffic to cluster (this could be easily done by using an Ethernet hub or specially configured Ethernet switch); and packet filter selects packets to be handled by given node and drops all other packets. Packet filter (later - PF) works only on external interface. Since it handles all cluster's incoming traffic (and that might be a lot of data), it must be as fast and efficient as possible. Partly because of this requirement it works at kernel level, that is inside the OS kernel. Due to the portability reasons and difficulties in debugging kernel parts, it makes sense to keep PF as simple as possible. Technically PF could be implemented as kernel module, which is preferable way, or as a patchset to the kernel. In proof-of-concept software, however, Packet Filter is implemented as a patch to FreeBSD *ipfw* packet filter; that patch adds another type of rule to ipfw due to the implementation simplicity reasons. The only configurable parameter of packet filter is a bitmask, which controls which traffic belongs to a given node. To explain how PF works, a typical client-proxy and proxy-server traffic exchange at IP level should be examined. In direct proxy configuration clients are configured explicitly to use web-cache, so clients send their requests directly to web-cache, so origin web server's address is not present at all in IP or TCP in client-proxy exchange. Here is a typical packet trace of such exchange taken with Unix tcpdump utility[26]:

```
20:23:33.198054        client.3716        >        cache.squid:        P
4215054594:4215055240(646) ack 4068770742 win 62943 (DF)
```

[26] In this output of tcpdump for better understandability real IP addresses of client and cache were replaced with words, and web-cache port number (usually 3128) was replaced with 'squid' word.

```
20:23:33.229614 cache.squid > client.3716: . ack 646 win 10336 (DF)
20:23:33.577894 cache.squid > client.3716: P 1:236(235) ack 646 win
10336 (DF)
20:23:33.734431 client.3716 > cache.squid: . ack 236 win 64240 (DF)
20:23:39.717226 client.3716 > cache.squid: P 646:1292(646) ack 236
win 64240 (DF)
20:23:39.717254 cache.squid > client.3716: . ack 1292 win 11628 (DF)
20:23:40.096954 cache.squid > client.3716: P 236:479(243) ack 1292
win 11628 (DF)
20:23:40.243789 client.3716 > cache.squid: . ack 479 win 63997 (DF)
20:23:45.380841 client.3716 > cache.squid: P 1292:1867(575) ack 479
win 63997 (DF)
20:23:45.380884 cache.squid > client.3716: . ack 1867 win 12920 (DF)
20:23:45.765031 cache.squid > client.3716: . 479:1939(1460) ack
1867 win 12920 (DF)
20:23:45.765051 cache.squid > client.3716: P 1939:1983(44) ack 1867
win 12920 (DF)
20:23:45.765683 client.3716 > cache.squid: . ack 1983 win 64240 (DF)
20:23:45.766740 cache.squid > client.3716: P 1983:3443(1460) ack
1867 win 12920 (DF)
20:23:45.952007 client.3716 > cache.squid: . ack 3443 win 64240 (DF)
20:23:45.957300 cache.squid > client.3716: P 3443:4903(1460) ack
1867 win 12920 (DF)
20:23:45.958509 cache.squid > client.3716: P 4903:5751(848) ack
1867 win 12920 (DF)
20:23:45.959087 client.3716 > cache.squid: . ack 5751 win 64240 (DF)
```

In case of transparent proxy users' web-browsers do not need to be configured explicitly; instead all traffic to 80th TCP port will be redirected by router to web-cache, and all packets' source and destination addresses will be left unmodified. The following tcpdump trace illustrates this scenario:

```
20:25:14.768189 client.3717 > webserver.http: S
4242166151:4242166151(0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
20:25:14.768234 webserver.http > client.3717: S
4192120992:4192120992(0) ack 4242166152 win 5840 <mss
1460,nop,nop,sackOK> (DF)
20:25:14.768426 client.3717 > webserver.http: . ack 1 win 64240 (DF)
```

```
20:25:14.768947 client.3717 > webserver.http: P 1:620(619) ack 1
win 64240 (DF)
20:25:14.768988 webserver.http > client.3717: . ack 620 win 6809
(DF)
20:25:15.149443 webserver.http > client.3717: P 1:230(229) ack 620
win 6809 (DF)
20:25:15.280527 client.3717 > webserver.http: . ack 230 win 64011
(DF)
20:25:16.395290 client.3717 > webserver.http: P 620:1239(619) ack
230 win 64011 (DF)
20:25:16.395329 webserver.http > client.3717: . ack 1239 win 8047
(DF)
20:25:16.774775 webserver.http > client.3717: P 230:459(229) ack
1239 win 8047 (DF)
20:25:16.882832 client.3717 > webserver.http: . ack 459 win 63782
(DF)
20:25:17.380955 client.3717 > webserver.http: P 1239:1858(619) ack
459 win 63782 (DF)
20:25:17.380999 webserver.http > client.3717: . ack 1858 win 9285
(DF)
20:25:17.762128 webserver.http > client.3717: P 459:688(229) ack
1858 win 9285 (DF)
20:25:17.884266 client.3717 > webserver.http: . ack 688 win 63553
(DF)
20:25:18.011027 client.3717 > webserver.http: P 1858:2477(619) ack
688 win 63553 (DF)
20:25:18.011056 webserver.http > client.3717: . ack 2477 win 10523
(DF)
```

Data exchange between web-cache and origin web-server is the same for direct and transparent proxy and typical packet exchange in this case looks like:

```
20:21:13.502129 cache.56318 > webserver.http: S
3932647754:3932647754(0) win 5840 <mss 1460,sackOK,timestamp
622921093 0,nop,wscale 0> (DF)
20:21:13.690110 webserver.http > cache.56318: S
1497256923:1497256923(0) ack 3932647755 win 57344 <mss 1460> (DF)
20:21:13.690159 cache.56318 > webserver.http: . ack 1 win 5840 (DF)
20:21:13.690684 cache.56318 > webserver.http: P 1:697(696) ack 1
win 5840 (DF)
```

```
20:21:13.884483 webserver.http > cache.56318: P 1:193(192) ack 697
win 58400 (DF)
20:21:13.884523 cache.56318 > webserver.http: . ack 193 win 6432
(DF)
20:21:13.884713 webserver.http > cache.56318: F 193:193(0) ack 697
win 58400 (DF)
20:21:13.885150 cache.56318 > webserver.http: F 697:697(0) ack 194
win 6432 (DF)
20:21:14.072419 webserver.http > cache.56318: . ack 698 win 58400
(DF)
```

The following figure illustrates all three possible types of data exchange for transparent and direct client-cache traffic and for cache-web-server exchange:

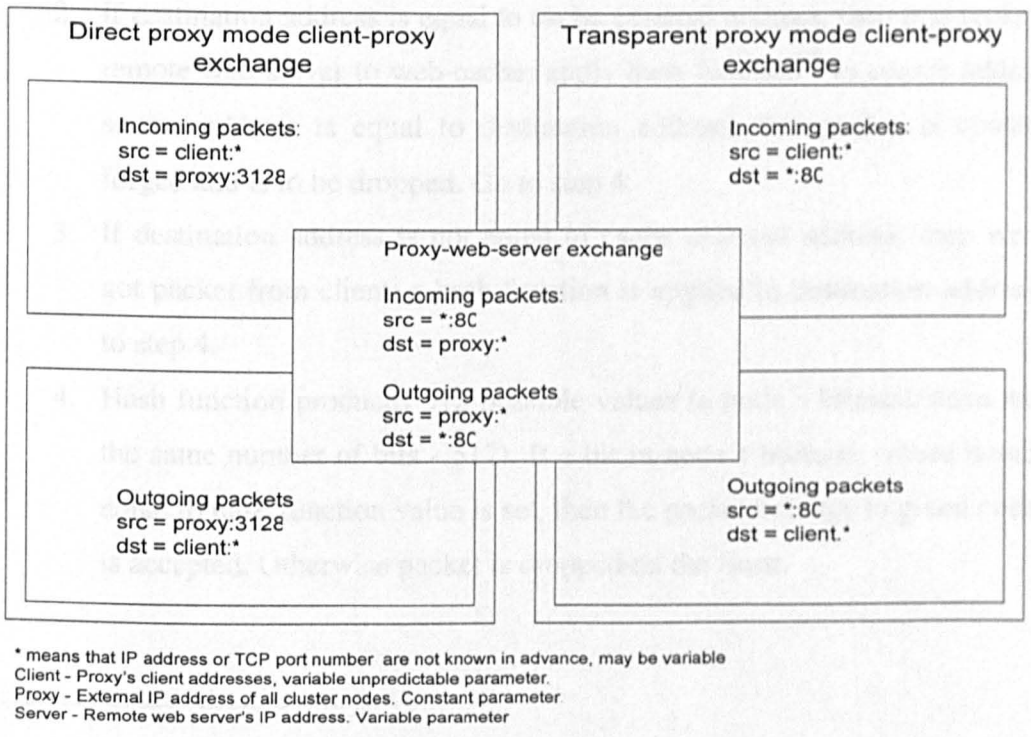


Figure 18. Possible traffic flows.

For the proposed cluster scheme to work, every node must have ability to use external interface *both* for client-cache and cache-web-server data exchange, so packet filter must provide ability to know that incoming packet is really intended for given node. In case of direct proxy it is impossible to determine which node has to handle incoming packet from web-user, because this packet would not contain any web-server address (see Fig.18 and sample traffic exchanges above for illustration). In case of transparent proxy however there is one common detail in both types of data exchange – IP address of origin web-server, so packet filter can use this detail in packet filter's algorithm as follows (see also Fig. 19).

1. PF analyzes only incoming traffic and of this traffic only destination and source IP addresses matter. This algorithm is applied to every incoming TCP packet.
2. If destination address is equal to cache external address, then it is reply from remote web-server to web-cache; apply hash function[27] to source address. If source address is equal to destination address, this packet is considered forged and is to be dropped. Go to step 4.
3. If destination address is not equal to cache external address; then we have got packet from client; a hash function is applied to destination address. Go to step 4.
4. Hash function produces 512 possible values (a node's bitmask have exactly the same number of bits - 512). If a bit in node's bitmask whose number is equal to hash function value is set, then the packet belongs to given node and is accepted. Otherwise packet is dropped on the floor.

[27] Hash function is quite simple and at present time is just a sum of IP address' octets modulo 512.

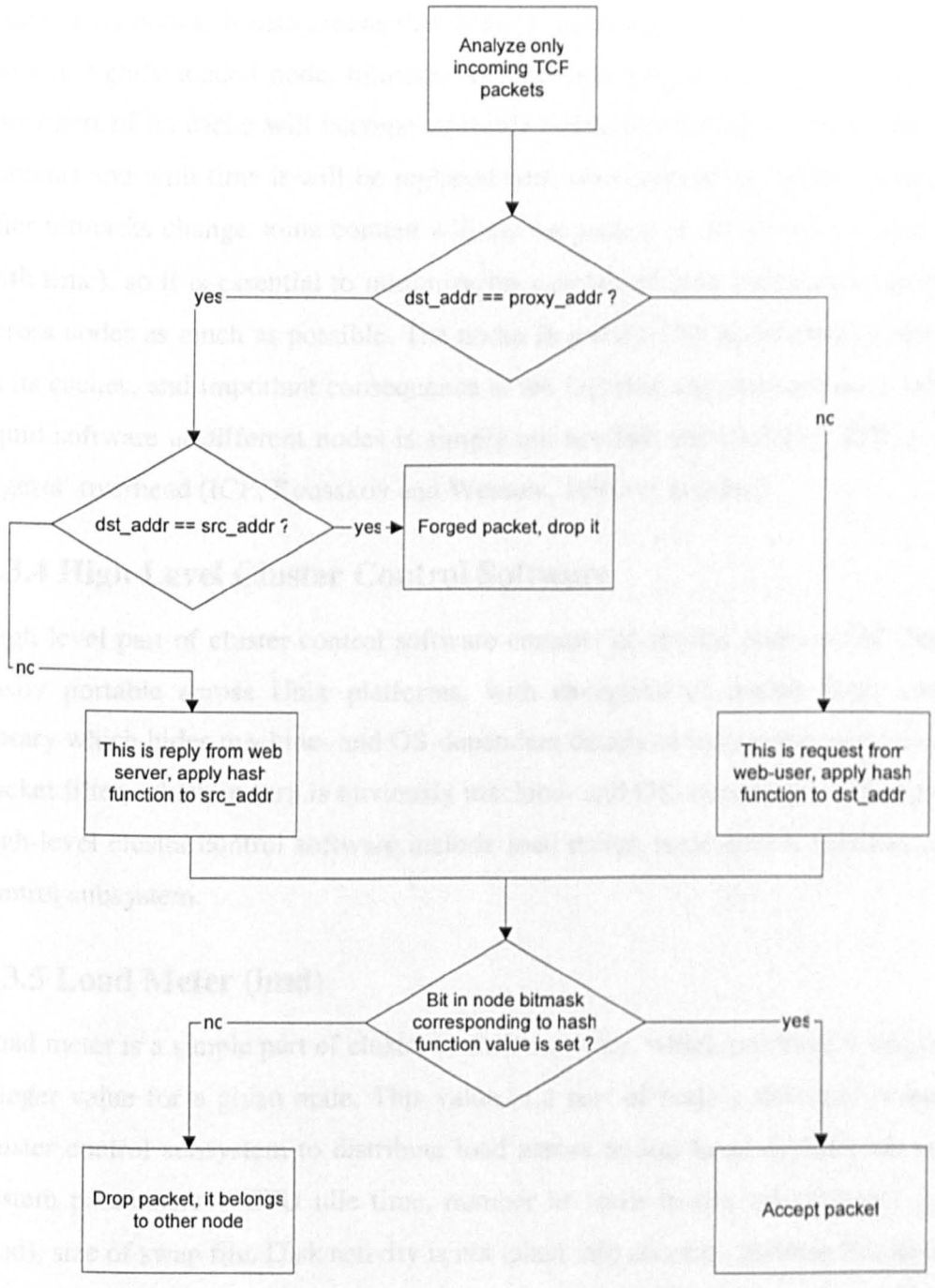


Figure 19. Packet Filter algorithm.

It follows from PF algorithm that each web-cache in cluster will serve its own set of remote web-sites, and these sets of sites will not overlap in nodes' disk caches, so total disk space capacity of web-caching cluster will be effectively a sum of disk space of its nodes. It also means that when load is distributed from heavily loaded node to lightly loaded node, bitmasks will be changed, so at heavily loaded node some part of its cache will become unusable (there will not be any requests to this content) and with time it will be replaced with new content; at lightly loaded node after bitmasks change some content will not be present at all (it will get into cache with time), so it is essential to minimize the number of load distribution operations across nodes as much as possible. The nodes in a cluster do not share any resources in its caches, and important consequence is the fact that any interoperation between squid software at different nodes is simply not needed; and therefore ICP or cache digests' overhead (ICP; Rousskov and Wessels, 1998) is avoided.

6.3.4 High Level Cluster Control Software

High level part of cluster control software consists of several parts; all of them are easily portable across Unix platforms, with exception of packet filter interface library which hides machine- and OS-dependant details of interaction with low-level packet filter, which in turn is obviously machine- and OS- dependant. Other parts of high-level cluster control software include load meter, node health monitor, cluster control subsystem.

6.3.5 Load Meter (lmd)

Load meter is a simple part of cluster control software, which produces a single load integer value for a given node. This value is a part of node's state and is used by cluster control subsystem to distribute load across nodes. Load depends on several system parameters – CPU idle time, number of tasks in run queue (Unix system load), size of swap file. Disk activity is not taken into account, because it is assumed that it will change proportionally to the CPU load. For simplicity purposes, all nodes should have identical configuration (that is CPU and RAM should be roughly the

same, number of hard drives must be exactly the same). Load of the node must be a very conservative value, it must not change fast (as it said earlier, every load redistribution leads to partial loss of cache content, and the more frequently the load of given node changes, the more is the chance that the load would need to be redistributed), so it can not only be based on current momentum values. In fact, load is recalculated every minute, and to calculate it, 60 momentum values for 60 last minutes are taken into account, so short-time surges in node's load will not affect significantly load value produced by load meter. In addition, load meter has built diagnostics routines which allow it to detect failures in other part of cluster software (healthd and clusterd) and safely reset and shutdown cluster software in case unrecoverable failure is detected.

6.3.6 Health Monitor (healthd)

Another integral part of cache control software is a health monitor. It checks every minute that web-cache software is running, that there are enough disk space, that allocated memory is not exceeding reasonable expectations, system load is within reasonable practical limits, and the node's hardware is working properly. Many of the hardware failures (like hard drive possible failure) may be diagnosed by analyzing log files or some kernel counters. Typical example is diagnostics delivered by SMART-enabled hard drives. In case of failure healthd safely shutdowns cluster software. Healthd functionality is partly duplicated in lmd (for reliability).

6.3.7 Cluster Control Subsystem (clusterd)

Cluster control subsystem is a heart of cluster control software. It receives events from other software parts (new node introduction, timer events, node hardware failure, unexpected connectivity loss with one of the nodes, uneven load distribution etc.), synchronizes cluster state using Cluster State Protocol (CSP). Every node keeps its own copy of cluster state. Cluster state is a table consisting of records each describing a particular node of the cluster:

1. *Node unique identifier* – 32-bit integer. Currently IP address of internal Ethernet interface is used for this purpose.
2. *Node status* – 32-bit integer. It can have several values including ACTIVE, PASSIVE, OFFLINE, FAILED, DEAD.
3. *Status update time* – two 32-bit integers describing the time when information about given node was last updated. This field is used to detect failed nodes which stop sending status information.
4. *Active flag* – 32-bit integer, which can have just two values – ACTIVE or PASSIVE. Only one node at a given time may have ACTIVE flag set. If there are no nodes in the cluster having this flag set, or there are two or more nodes with this flag, an *election* process is initiated.
5. *Node load* – 32-bit integer describing integral load of the given node which is reported by load meter.
6. *Node bitmask* – 512-bit integer (actually a set of 16 32-bit integers) describing which part of partitioned web objects space a given node is responsible for.
7. Node with Active flag set has a 32-bit integer containing the time of last load redistribution caused by load unevenness.

One node of the cluster has ACTIVE flag set and only this node can take load redistribution decisions and can send commands to other nodes. If active node is lost, or there are two active nodes[28], active node election process is initiated, and node with minimum node identifier will be selected as new ACTIVE node. All nodes periodically report their state to other nodes via CSP.

[28] Every node keeps its own copy of cluster state (states of all nodes in the cluster), so this situation can be trivially detected.

There are several possible states of the nodes:

1. INIT state. This is initial state of the node after clusterd start-up.
2. INTRO state. Node start-up is finished and node is ready to join the cluster. For several seconds (this value is random in some range to minimize probability of simultaneous joins of the nodes during simultaneous start-up of multiple nodes, at the time of writing this value is from 5 to 60 seconds) node is quietly listening to CSP messages from other nodes. If there are no messages, then node moves to ACTIVE state, initializes bitmask with all ones and becomes the first node in the cluster. If there are CSP messages from other nodes, then the node assumes that it is not the first one in the cluster and goes to PASSIVE state and initializes all bits of its bitmask to zero.
3. ACTIVE state. Node in this state controls the whole cluster. When it detects some specific conditions (listed below) it goes to ACTIVE_CMD mode and issues a set of commands to other cluster nodes using CSP. Every 200 ms it goes to ACTIVE_TX state. Every 50 ms it goes to ACTIVE_RX state.
4. ACTIVE_TX state. This is a short-term state in which node emits its state to other nodes via CSP and immediately goes back to ACTIVE state.
5. ACTIVE_RX state. This is a short-term state in which node reads state information from other cluster nodes, updates its internal cluster state table and immediately goes back to ACTIVE state.

6. ACTIVE_CMD state. In this state node generates a redundant (due to the possible UDP datagram loss) sequence of commands[29] to other nodes and sends these commands, after that the node goes back to ACTIVE state.
7. PASSIVE state. Node in this state only executes commands from node in ACTIVE state. Every 200 ms it goes to PASSIVE_TX state. Every 50 ms it goes to PASSIVE_RX state.
8. PASSIVE_TX state. This is a short-term state in which node emits its state to other nodes via CSP and immediately goes back to PASSIVE state.
9. PASSIVE_RX state. This is a short-term state in which node reads state information from other cluster nodes, updates its internal cluster state table and immediately goes back to PASSIVE state.
10. ELECTION state. When any node detects either loss of connection with ACTIVE node or presence of two or more ACTIVE nodes, it goes to ELECTION state and initiates an *election process* (described below).
11. OFFLINE state. Every node can be put offline by cluster administrator.
12. FAILED state. When node's software detects internal inconsistency (see below) and/or software or hardware failure, it switches the node to FAILED state.
13. DEAD state. This state is a virtual state of node which has just disappeared without any signals of failure.

[29] At present time commands are just repeated 3 times.

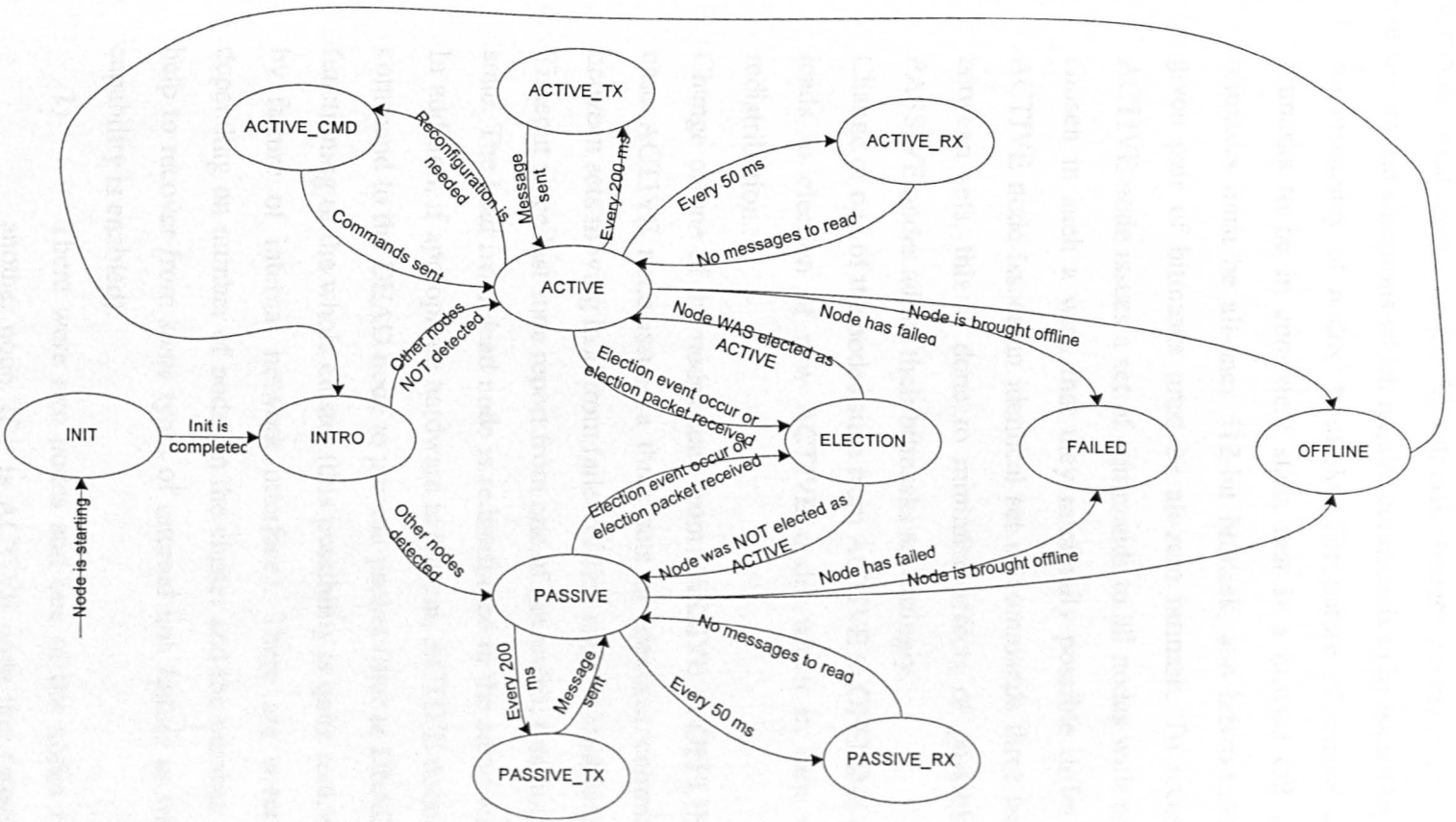


Figure 20. Cluster state diagram.

ACTIVE node controls the behaviour of the cluster, maintains cluster state consistency and redistributes load among nodes. All bitmask changes are made by ACTIVE node commands. There are several situations which require commands to be issued by ACTIVE node:

1. Inconsistency of nodes' bitmasks. The nature of cluster architecture requires all bitmasks to be in consistent state, that is a bitwise OR operation of all nodes' bitmasks must be all-ones 512-bit bitmask, and bitwise AND operation of every given pair of bitmasks must be all-zero bitmask. To recover from this situation ACTIVE node issues a set of commands to all nodes with new bitmasks, which are chosen in such a way, that they minimally possible differ from initial bitmasks. ACTIVE node issues an identical set of commands three times with a 0.5 s pause between sets, this is done to minimize effects of possible UDP datagram loss. PASSIVE nodes adjust their bitmasks accordingly.
2. Change of one of the nodes state from ACTIVE to OFFLINE or FAILED. This event leads to election of new ACTIVE node, which in turn will perform bitmasks redistribution.
3. Change of one of the nodes state from PASSIVE to OFFLINE or FAILED. In this case ACTIVE node issues a three sets of identical commands with 0.5 s delay between sets moving load from failed/offline node to least loaded nodes.
4. Timeout since last state report from one of the nodes; this node is considered in dead state. The load from dead node is redistributed in the same way as from failed node. In addition, if appropriate hardware is present, ACTIVE node must issue a poweroff command to the DEAD node to prevent packet filter at DEAD node from disrupting functioning of the whole cluster (this possibility is quite real, e.g. this may be caused by failure of internal network interface). There are several recovery scenarios depending on number of nodes in the cluster and the number of failed nodes, which help to recover from *some* types of internal link failure as well (provided poweroff capability is enabled):
    1) There were two nodes and one of the nodes fails from the point of another node. If it is ACTIVE node that failed, PASSIVE node will discover loss of communication with ACTIVE node and initiate an election process. If it is PASSIVE node that failed, then ACTIVE node will detect it and turn PASSIVE node off. If it is an internal link which

has failed, then both ACTIVE and PASSIVE nodes will detect failures of each other, but since dead node discovery timeout at ACTIVE node is deliberately chosen twice as shorter than at PASSIVE mode, ACTIVE node will switch PASSIVE node off and cluster will still continue to work with just one node.

2) If any node looses connectivity with more than one node it assumes that its internal link has failed, and switches itself to FAILED state (resetting packet filter as well). Therefore if there were many nodes and connectivity between *all of them* is lost, then the whole cluster goes down. If just one node of many fails, then if it was ACTIVE node, election is started. If it was PASSIVE node, then ACTIVE node redistributes load from DEAD node to other nodes using redundant set of CSP commands.

5. Too big difference between loads of least loaded node and most loaded node. If this difference exceeds some threshold (3 times by default), then load is partially redistributed from most loaded node to least loaded node by issuing redundant set of commands[30] to a pair of nodes. Only 2 nodes participate in load redistribution, only one bit of the bitmask is moved from one node to another at a time, and the frequency of such redistribution is limited by no less than 1 per minute. This is done because every load redistribution effectively discards part of web-cache storage.

Following figures illustrate load redistributions caused by one node failure and by several load redistribution decisions due to the unevenness of load among nodes.

---

[30] A set of commands will be repeated a few times (currently 3 times).

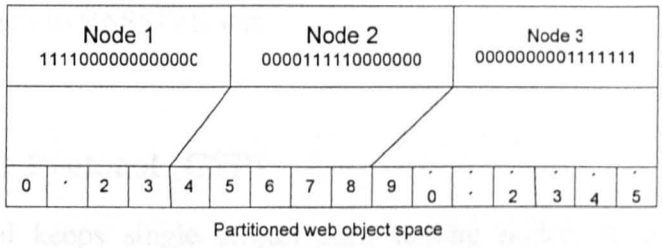


Figure 21. Bitmasks: stable cluster state.

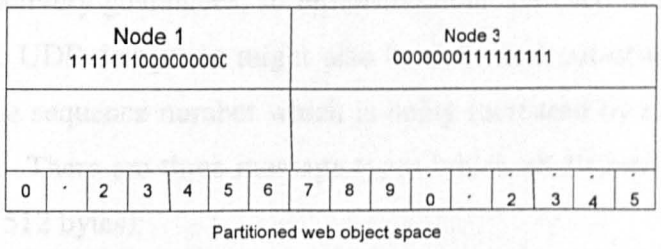


Figure 22. Bitmasks: after node 2 failure.

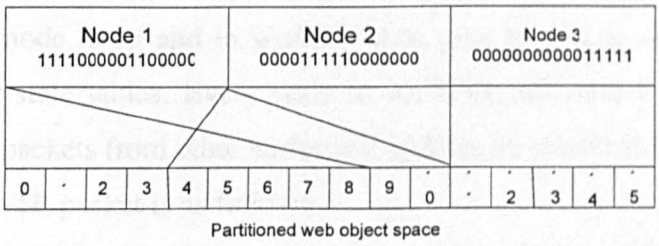


Figure 23. Bitmasks: after load redistribution.

Election process is initiated when one of the nodes sends a series of ELECTION CSP packets. Every node which is currently in PASSIVE or ACTIVE state switches itself to ELECTION state. If the node in ELECTION state receives ELECTION packet from the node with bigger node identifier, it answers with a series of ELECTION packets, otherwise it does not send anything. Process continues until the node with least node identifier answers.

That node becomes a new ACTIVE node after some timeout is expired. Other nodes in ELECTION state switch to PASSIVE state.

6.3.8 Cluster State Protocol (CSP)

Cluster state protocol keeps single cluster state among nodes. It somewhat resembles protocol used in Token Ring (Cisco book; 802.5) LANs, but without any unneeded complexities of Token Ring. CSP is based on UDP and by default uses port 7777. UDP does not provide reliable delivery guarantees, so retransmissions are used to increase probability of messages delivery. UDP datagrams might also be delivered out-of-order, so every CSP packet contains unique sequence number which is being increased by one for every packet sent by a given node. There are three message types which all fit into single UDP packet (with data payload of 512 bytes):

1. STATE packet which is emitted by every node in ACTIVE_TX and PASSIVE_TX states every 200 ms by default. This packet has source address of sending node and destination address is set to limited broadcast address. The purpose of this packet is to signal that node is up and in working state plus to update node's state in other nodes' cluster state tables. Every node in ACTIVE_RX and PASSIVE_RX states reads STATE packets from other nodes and updates its cluster state table accordingly. Format of STATE packet is as follows:
    1) Packet type – 32-bit integer equal to 3 symbols "STA" in C language notation (with terminating zero byte).

2) Node state – 32-bit integer. Node state of originating node, possible values: ACTIVE, PASSIVE, OFFLINE, FAILED, DEAD[31].
3) Sequence number – 32-bit integer.
4) Node load – 32-bit integer – load value from lmd.
5) Bitmask – 512-bit integer (64 bytes).
6) MD5[32] checksum of 1-5 fields (16 bytes)

2. ELECTION packet is generated by a node in ELECTION state when one of the specific conditions was discovered. It has the same format as STATE packet, only Packet Type is set to "ELE" in C language notation.
3. COMMAND packet is generated by ACTIVE node in ACTIVE_CMD state when specific conditions described above occur in the cluster. The source address of this packet is address of ACTIVE node, destination address of this packet is address of specific node (not broadcast address). It has the following fields:
   1) Packet type – 32-bit integer equal to "CMD" in C language notation.
   2) Sequence number – 32-bit integer
   3) New bitmask for a given node – 512-bit integer.
   4) MD5-checksum of 1-3 fields.

[31] While no node will ever send a packet with DEAD state, a packet with FAILED state is possible (for example, when failure of web-caching software at the node was detected).

[32] Message Digest 5 – cryptographically strong hash function, which operates on arbitrary number of bytes and returns 16 byte value.

6.3.9 Software Implementation Details

High-level cluster control software is written in C++ (around 10 000 lines of source code) and is implemented as several userspace daemons: lmd (Load Meter), healthd (Health Monitor), clusterd (cluster state protocol and control subsystem), that interact with each other via Unix sockets. There are also a number of command-line utilities which are used for cluster start-up and monitoring. Packet Filter is currently implemented as a patchset to FreeBSD 4.10 ipfw packet filter. This patchset adds a couple of sysctl variables, so it is possible to set or get these bitmasks via generic *sysctl*[33] interface (via PF interface library, which hides implementation details). Patchset is implemented in C language (around 100 lines of source code). At its current state cluster management software is quite simple and does not spend any significant system resources when running. All configurations are performed via single configuration file, which contains only a handful of variables, including cluster node number, external and internal IP addresses. All nodes administrative operations (start node, move node offline, monitor cluster state, get node statistics) are performed by using a single script, which is named 'node'. By default cluster software installs itself to /usr/local/cluster directory. Configuration of nodes is almost the same (only a few parameters change from node to node), so it is possible to create a complete image of one node and then quickly duplicate it to other nodes, so multiple OS, squid and cluster software reinstalls are not needed (of course after this duplication some parameters must be changed), which greatly simplifies cluster administration.

[33] Standard UNIX interface to read and modify kernel variables from userspace program.

6.4 Experimental Results

3-node cluster was used for experimental purposes. All nodes had almost the same configuration – 1.6GHz PIV CPU, 512MB RAM, single 60GB hard drive. Fourth computer was used in server role, and fifth computer was used in client role. Since this is proof-of-concept software, most of experiments only involved light loads, any benchmarking was not performed. All nodes with exception of server computer, which had RedHat Linux 7.3, were running FreeBSD 4.10-RELEASE operating system with packet filter patchset. Squid version was 2.5.STABLE6. Memory cache was set to 128MB, disk cache to 1GB. For both external and internal networks Fast Ethernet switched network was used. The whole cluster was controlled from laptop computer attached to the internal network.

A series of tests has been performed with the main goal to confirm that software works as expected and that assumptions made when designing software were actually true.

1. Functioning of the packet filter. The idea of cluster with common IP and MAC address at external interface is actually not new (Vaidya and Christensen, 2001) and is proven to be working, however in proposed web-caching cluster external interface is used not only for accepting and serving user's requests, but also for performing web object retrieval operations from remote web-servers. Therefore packet filter must be built in such way so user requests *and* corresponding fetch operations must be served by the same node. An experiment with three cluster nodes has been performed with success.
2. Performance impact of packet filter. In proposed cluster scheme every node of the cluster must handle all incoming traffic going to the whole cluster, so performance impact of filtering may be significant. To estimate this negative impact an experiment was conducted. One relatively slow 366MHz computer running FreeBSD 4.10-RELEASE was tested with and without packet filter running. Such a slow computer was selected because the impact of packet filter on relatively modern computer with Fast Ethernet would not be easily visible. In both cases a stream of 8500 packets per second was directed to that computer. In case without packet filter

system time was about 21% of total CPU power, in case with packet filtering system time was about 26%. Since 8500 packets per second with 1000B average packet size is around 8MBps, which is close to practical maximum bandwidth of Fast Ethernet , we can interpolate and conclude that even non-optimized packet filter running on slow 366MHz machine with 66MHz system bus can handle up to 400Mbps link (provided that test machine has Gigabit Ethernet interface). Better hardware and better packet filter will allow a single PC computer with price below €1000 to handle up to several gigabits of traffic.

3. Cluster control software relies on the correctness of bitmasks in its work; any inconsistencies in bitmasks could lead to partial or even complete failure of the cluster. Therefore bitmask consistency recovery procedures are implemented in cluster software. Testing results have shown that any bitmask inconsistency is fixed in 2 seconds or less.
4. Another set of experiments was performed to estimate how fast CSP reacts to various events. A 3-node cluster was used for this purpose. Several types of events were simulated including web-caching software (Squid) failure, excessive load and instantaneous disappearance of one of the nodes (using network disconnect) both in active and passive mode. It was determined that for 3-node cluster with default parameters any recoverable failure not relating to active monitor will be corrected in time period under 2 seconds. Any failures concerning active monitor will be corrected in time period up to 5 seconds, because in this case an election process would be required.

6.5 Advantages and Limitations

Proposed cluster solution to the problem of building high-performance fault-tolerant web-cache has significant advantages over most of the other proprietary and open solutions. First, it is cheap and has a good price-performance ratio. Second, it is highly scalable and performance of cluster can be increased on demand by adding another cheap node without

need to replace or even stop the whole cluster. Third, the cluster is built from widely available blocks, nothing special is required, and any additional equipment like fancy Layer-4 switches are simply not needed. All cluster functions are completely software-based. Fourth, the reliability of the whole cluster is high enough, so cluster will recover from most of the failures in mere seconds. Fifth, cluster's performance may range from 20 Mbps (in 2-node configuration with low-end nodes) to whole 622Mbps ATM link (with 16 midrange nodes). Sixth, cluster operation is completely automatic, it does not require any immediate administrator intervention, most of the problems will be overridden by cluster software and may be fixed by support personnel later in due time. Cluster software is quite simple to use, and software running on the nodes does not require any major modifications, so there is no need for extensive training of system administrators.

All these advantages significantly outweigh limitations of the system; however these limitations should be taken into account when designing and operating web-caching cluster. First, all cluster's logic depends on reliability of network connections (especially internal network used for cluster state protocol), so either duplicate links are required or some extensive monitoring of network devices' operation. There are built-in recovery mechanisms from failure of internal network card, but when internal Ethernet switch goes down, it effectively means the whole cluster goes down. Off course, there exist other network technologies, such as FDDI and Token Ring, which have reliable delivery guarantees and built-in self-healing capabilities. However, these networking technologies are much more expensive and not as widely used as Ethernet, and networking hardware for these technologies is usually not readily available. Duplication of internal Ethernet connection will provide better reliability than single FDDI or Token Ring network, at the same time

requiring only minor modifications to Cluster State Protocol *and* still being cheaper than FDDI or Token Ring[34].

Second, cluster only works with transparent scheme; to implement direct caching scheme it would be needed to add a third network interface to each of the nodes and use more complex packet filter on 2 interfaces. However, due to the practical difficulties in using direct scheme, this limitation is rather inexistent. Third, cluster software does not recover from all possible failures. There are some very unlikely situations where one failed node can significantly degrade cluster's robustness, e.g. when usermode cluster control software will fail and the load will be moved to another node, but kernel-mode packet filter at failed node will continue to operate disrupting all requests to sites corresponding to its bitmask. Recovering from these unlikely situations will require more complex cluster control algorithms and additional hardware, e.g. power control devices[35], so failed node could be turned off by cluster software. However, these improvements will be evolutionary and will not significantly change proposed cluster structure.

Packet Filter is a possible bottleneck of the proposed system, it must handle a possibly very high packet rate, so both efficient hardware and efficient processing is required. E.g., assuming overall available bandwidth of 622Mbps and average packet size of 512 bytes, the number of arriving packets would be approximately 160 000 packets per second. Older network cards generate an interrupt for every incoming packet, causing context switch, which is a quite expensive operation in Unix (CPU cache must be flushed, registers saved,

[34] Well, there is no support for Token Ring networking in FreeBSD anyway; it seems nobody of FreeBSD developers and users is interested in this networking technology.

[35] For example, American Power Conversion (APC) has a suitable product (MasterKey).

etc.), so using these old cards may lead to extremely high interrupt rates and extremely high load, possibly making the whole system unresponsible. Happily, most of the Gigabit Ethernet cards are quite intelligent devices, which can even handle checksum processing and generate interrupts for a batch of packets, and not for every packet. Some cards support *polling* mode, so no interrupts are generated at all when packet arrives, but operating system itself polls network card a few hundred times of second for new packets. Some cards are programmable, so it becomes possible to implement packet filter inside network card, completely offloading CPU of the node itself. These optimizations might be applicable to production systems; however for the proof-of-concept software these optimizations are not needed.

6.6 Summary

Design goals of proposed cluster implementation are the best possible price/performance ratio, high performance (up to 500Mbps), scalability, extendibility and fault tolerance. Practical implementation of web-caching cluster would require only widely available building blocks such as PCs and Ethernet switches. Proposed approach to web-caching cluster is completely software based, it requires minimal training of support personnel and minimal effort to setup and maintain the cluster. Particular cluster implementation requires several nodes with two network interfaces each, at external interface all nodes have the same IP and MAC addresses, at internal interface nodes have different IP and MAC addresses. The consequence of this architectural decision is that every node gets a full copy of incoming traffic of the cluster. Internal interface is used for inter-node communications. Each node has a copy of cluster software, which consists of several parts, mainly high-level control software and low-level packet filter. High-level control software performs load-balancing and recovery from faults, state of the cluster is synchronised using a new original Cluster State Protocol (CSP). Low-level packet-filter performs selection of incoming traffic belonging to given node. Proof-of-concept software has been written to illustrate functionality of the proposed approach to build web-caching clusters. Experimental results

have shown that software basically works and is capable of meeting stated above design goals. Proposed web-caching cluster has probably the best price/performance ratio among web-caches capable of handling from 100Mbps to 500Mbps of HTTP traffic with the same level of reliability and fault-tolerance.

CHAPTER 7

CONCLUSION

The goal of building high-performance web-cache with best possible price/performance ratio can be achieved only using cluster approach. Increasing performance of a single caching unit by utilizing specialized filesystems and network stacks or highly-efficient multithreaded web-caching software can provide very significant performance gains, but still can not solve severe problems caused by persistent storage shortcomings. The only scalable way of building web-cache is to utilize several caching units organized into cluster. Proposed cluster scheme uses software-based approach; it does not require any additional hardware (except caching units themselves). It was demonstrated that relatively simple software solution can achieve excellent price/performance ratio, good performance and reliability, fast fault recovery times, the best possible disk cache utilization, which uses widely available OS and caching software and requires minimal effort to setup and maintain.

7.1 Achievement

In chapter 1 a few criteria were proposed which are to be used to estimate the success of thesis. Most of the answers are definitely positive, and remaining ones are supposedly positive.

- *Which benefits does the proposed clustering scheme provide in comparison to other methods of building high-performance web-caches?*

Proposed solution has supposedly the best possible price/performance ratio and is completely based and uses existing software and hardware, proposed cluster is to be constructed from cheap building blocks. One of the main advantages of proposed solution is that it is intended from the very beginning to be used for web-caching purposes and takes into account web-caching properties and characteristics, which allows to achieve higher performance, better reliability and failover capabilities.

- *Does the proposed web-cache cluster meet its design goals*?

Proposed cluster fully meets design goals stated in paragraph 6.1. One of the main design goals was to achieve the best possible price/performance ratio, and that task is accomplished as well as all secondary tasks.

- *Does it satisfy performance and reliability requirements*?

As stated above web-caching cluster could be constructed from up to 32 nodes, achieving throughput of 1Gbps of HTTP traffic, which should be enough for most deployments. If higher performance is needed, several clusters can be combined using e.g. DNS Round-Robin Distribution. Nodes' failures are detected automatically, and failed node is put out of operation in few seconds completely automatically.

- *Does it provide required load-balancing and failover capabilities*?

Cluster control software performs automatic load balancing, providing protection from 'Slashdot effect' (Adler, 1999) and other traffic anomalies. Failover capabilities are provided.

- *Are there any networks where proposed web-cache cluster would be really useful*?

Any network which has high-speed and relatively expensive communication links to the Internet (e.g. international link) will benefit from deployment of proposed web-caching cluster. Web-object retrieval times will decrease significantly, and bandwidth will be used more efficiently.

- *What is the expected location in the network of web-caching cluster*?

Optimal place for web-caching cluster is next to the backbone links. Ideally all HTTP traffic should go via web-cache, which can be only achieved by using transparent web-caching.

- *Is the proposed approach practically feasible*?

The answer is definitely yes. Proposed solution does not require significant investments, existing servers can be used without modifications, and maximum performance of web-caching system can be increased at any time by adding new caching nodes to running cluster. Existing web-caching system of co-operating web-caches can be trivially converted to web-caching cluster.

- *Can it be implemented in real network in such a way that any significant effort from system administrators and end users would not be required*?

The answer is yes, cluster software is installed and configured trivially, it uses widely available software, so support personnel do not need to spend a lot of time learning new software. From user's point of view the cluster is invisible, so no modifications or reconfigurations of users' computers are required.

7.2 Future Work

Web-caching performance requirements will certainly increase in the following few years. Proposed web-caching cluster scheme is without any doubt completely working and could be useful in real life. It was shown above that it potentially meets all requirements such as scalability, reliability, performance levels, load redistribution and failover capabilities. However proof-of-concept code is far from production quality and it will certainly require a significant effort to make it suitable for real web-caching purposes. Web-caching clusters must be capable of running without intervention for days, weeks and months, so cluster control software must be relatively bug-free and proven to be reliable. To achieve that thorough extensive testing will be needed. There are numerous configuration parameters which must be tuned to provide better load balancing and failover capabilities which will help to achieve better utilization of cluster hardware and to get better bandwidth savings. Production-quality cluster control software would certainly contain better configuration and monitoring tools, probably Graphical User Interface and in any case as fast as possible packet filters. First real-life deployment of cluster software would for sure teach valuable lessons what should be improved.

Proposed cluster state protocol has some limitations which may be overridden by using more sophisticated algorithms. By improving CSP (e.g. by adding to it support of multiple internal network interfaces as it is done in Totem Redundant Ring Protocol (Koch et al, 2002)) it would become possible to recover even from internal network failures. Load distribution methods incorporated into current version of software are not perfect, because one of the primary goals of proof-of-concept software was to keep it as simple and easy to understand as possible, so load redistribution is one of the areas which could be significantly improved to provide more even distribution of load across nodes and to minimize the number of load distributions.

The maximum number of nodes in web-caching cluster is limited by 32 nodes. Therefore maximum throughput of the whole cluster is limited by some value which might be not adequate for some situations. This limitation may be currently fixed by using a set of independent clusters with DNS-based or router-based round-robin load distribution across them. However it is also possible to modify cluster state protocol, so it would support more complex situations than current version of CSP, such as cluster of clusters.

REFERENCES

(802.5)

IEEE 802.5 Token-Ring Access Method.

http://standards.ieee.org/getieee802/download/802.5-1998.pdf

(Adler, 1999)

Stephen ADLER.

The Slashdot effect - an analysis of three Internet publications. http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html, 1999.

(Aghdaie and Tamir, 2001)

Navid AGHDAIE and Yuval TAMIR.

Client-Transparent Fault-Tolerant Web Service.

In Proceedings of the IEEE International Performance, Computing, and Communications Conference, Phoenix, AZ USA , April 2001.

(Agrawal et al, 2001)

Dakshi AGRAWAL, James Giles, and Dinesh C. VERMA.

On the Performance of Content Distribution Networks.

In Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems, Orlando, Florida USA, July 2001.

(Amer et al, 2002)

Ahmed AMER, Darrell D.E. LONG, Randal C. BURNS.

Group-based Management of Distributed File Caches.

Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), 2002.

(Andrews et al, 2002)

Matthew ANDREWS, Bruce SHEPHERD, Aravind SRINIVASAN, Peter WINKLER, and Francis ZANE.

Clustering and Server Selection using Passive Monitoring.

In Proceedings of the IEEE Infocom 2002 Conference, New York City , June 2002. IEEE.

(Apache)

Apache HTTP Server project.

http://httpd.apache.org

(Aversa and Bestavros, 2000)

Luis AVERSA and Azer BESTAVROS.

Load Balancing A Cluster Of Web Servers Using Distributed Packet Rewriting.

In Proceedings of the IEEE International Performance, Computing, and Communications Conference, Phoenix, AZ USA , February 2000.

(Baker and Moon, 1999)

Scott M. BAKER and Bongki MOON.

Distributed cooperative Web servers.

In Proceedings of the 8th International WWW Conference, Toronto, Canada,

http://www8.org/w8-papers/2a-webserver/distributed/distributed.html

May 1999.

(Barford and Crovella, 1999)

Paul BARFORD and Mark E. CROVELLA.

Measuring Web performance in the wide area.

Performance Evaluation Review, August 1999.

http://www.cs.bu.edu/faculty/crovella/paper-archive/wawm-per.ps

(Bent and Voelker, 2002)

Leeann BENT, Geoff VOELKER.

Whole Page Performance.

7th International Workshop on Web Content Caching and Distribution. Boulder, Colorado. August 14-16, 2002.

(Breslau et al, 1999)

Lee BRESLAU, Pei CAO, Li FAN, Graham PHILLIPS, and Scott SHENKER.

Web caching and Zipf-like distributions: Evidence and implications.

In Proceedings of the INFOCOM '99 conference, March 1999.

http://www.cs.wisc.edu/~cao/papers/zipf-like.ps.gz

(Brewington and Cybenko, 2000)

Brian E. BREWINGTON and George CYBENKO.

How dynamic is the web?

In Proceedings of the 9th International WWW Conference, May 2000. http://www9.org/w9cdrom/264/264.html

(BSD license)

The FreeBSD Copyright.

http://www.freebsd.org/copyright/freebsd-license.html

(Borcea et al, 2002)

Christian BORCEA, Deepa IYER, Porlin KANG, Akhilesh SAXENA, Liviu IFTODE. Cooperative Computing for Distributed Embedded Systems. Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), 2002.

(Bourke)

Tony BOURKE.

Server Load Balancing.

O'Reilly and Associates.

(Bunt et al, 1999)

Richard B. BUNT, Derek L. EAGER, Gregory M. OSTER, Carey M. WILLIAMSON.

Achieving Load Balance and Effective Caching in Clustered Web Servers. The 4th International Web Caching Workshop. San Diego, California, March 31 - April 2, 1999.

(Cardellini et al, 2003)

Valeria CARDELLINI, Michele COLAJANNI, Philip S. YU.

Request Redirection Algorithms for Distributed Web Systems.

IEEE Transactions On Parallel and Distributed Systems, Vol.14, No. 4, April 2003.

(Casalicchio and Colajanni, 2001)

Emiliano CASALICCHIO and Michele COLAJANNI.

A client-aware dispatching algorithm for Web clusters providing multiple services. In Proceedings of the 10th International WWW Conference, Hong Kong, May 2001. http://www10.org/cdrom/papers/pdf/p434.pdf

(Challenger et al, 1999)

Jim CHALLENGER, Arun IYENGAR, and Paul DANZIG.

A scalable system for consistently caching dynamic Web data. In Proceedings of IEEE INFOCOM'99, March 1999. http://www.research.ibm.com/people/i/iyengar/infocom2.ps

(Chen and Zhang, 2002)

Songqing CHEN, Xiaodong ZHANG.

Detective Browsers: A Software Technique to Improve Web Access Performance and Security.

7th International Workshop on Web Content Caching and Distribution. Boulder, Colorado. August 14-16, 2002.

(Chen and Zhang, 2002)

Songqing CHEN, Xiaodong ZHANG.

Adaptive and Virtual Reconfigurations for Effective Dynamic Job Scheduling in Cluster Systems.

Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), 2002.

(Chen et al)

Mao CHEN, Jaswinder Pal SINGH, Andrea LAPAUGH.

Subscription-enhanced Content Delivery.

(Cherkasova and Karlsson)

Ludmila CHERKASOVA, Magnus KARLSSON.

Scalable Web Server Cluster Design with Workload-Aware Request Distribution Strategy WARD.

(Chi et al, 2003)

Chi Hung CHI, HongGuang WANG, William KU.

Proxy-Cache Aware Object Bundling for Web Access Acceleration.

Web Content Caching And Distribution: Proceedings Of The 8th International Workshop, IBM T.J. Watson Research Centre, Hawthorne, New York, USA, September 29 - October 1st, 2003.

(Chung and Kim, 2001)

Ji Yung CHUNG and Sungsoo KIM. Efficient Load Balancing in a Heterogeneous Web Server Cluster.

In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada , June 2001.

(Cisco cache)

Cisco Cache Engine.

http://www.cisco.com/go/cache/

(Cisco handbook)

Cisco Internetworking Technologies Handbook: Token Ring.

http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/tokenrng.htm

(Cormack, 1998)

Andrew CORMACK.

Experiences with Installing and Running an Institutional Cache.

3rd International WWW Caching Workshop. University of Manchester. Manchester, England, June 15-17 1998.

(Cournier et al, 2002)

Alain COURNIER, Ajoy K. DATTA, Franck PETIT, Vincent VILLAIN. Snap-stabilizing PIF algorithm in arbitrary networks.

Proceedings of 22nd International Conference on Distributed Computing Systems, 2002. Page(s): 199- 206.

(Danzig, 1998)

Peter DANZIG.

NetCache Architecture and Deployment.

3rd International WWW Caching Workshop, Manchester, England, 1998.

http://www.netapp.com/technology/level3/3029.html

(Datta et al, 2002)

Anindya DATTA, Kaushik DUTTA, Helen THOMAS, Debra VANDERMEER.

A Proxy-Based Approach for Dynamic Content Acceleration on the WWW. Proceedings of Fourth IEEE International Workshop On Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS 2002). 26–28 June 2002. Newport Beach, California.

(Davison, 2001)

Brian D. DAVISON.

Assertion: Prefetching With GET is Not Good.

Proceedings of the Sixth International Workshop on Web Caching and Content Distribution. Boston University, Boston, Massachusetts, USA. June 20-22, 2001.

(Davison, 1999)

Brian D. DAVISON.

A Survey of Proxy Cache Evaluation Techniques.

The 4th International Web Caching Workshop. San Diego, California, March 31 - April 2, 1999.

(Davison) Brian D. DAVISON.

Web caching and content delivery pages.

http://www.web-caching.com

(Doyle et al)

Ronald P. DOYLE, Jeffrey S. CHASE, Syam GADDE, Amin M. VAHDAT.

The Trickle-Down Effect: Web-caching and server request distribution.

(Du and Subhlok, 2002)

Ping DU, Jaspal SUBHLOK.

Evaluation of Performance of Cooperative Web Caching with Web Polygraph. 7th International Workshop on Web Content Caching and Distribution. Boulder, Colorado. August 14-16, 2002.

(ESI)

Edge Side Includes. http://www.edge-delivery.org

(Feenan et al, 2002)

James FEENAN, Patrick FRY, and Ming LEI.

Clustering Web Accelerators.

Proceedings of Fourth IEEE International Workshop On Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS 2002). 26–28 June 2002. Newport Beach, California.

(FreeBSD)

The FreeBSD project.

http://www.freebsd.org

(Gadde, 1997)

Syam GADDE, Michael RABINOVICH, and Jeff CHASE.

Reduce, reuse, recycle: An approach to building large Internet caches.

In Proceedings of the HotOS '97 Workshop, May 1997. http://www.cs.duke.edu/ari/cisi/crisp-recycle.ps

(Ganger)

Gregory R. GANGER and Yale N. PATT.

Soft Updates: A Solution to the Metadata Update Problem in File Systems. Department of EECS, University of Michigan. http://www.ece.cmu.edu/~ganger/papers/CSE-TR-254-95/CSE-TR-254-95.ps

(GNU license)

GNU General Public License.

http://www.gnu.org/copyleft/gpl.html

(Guo et al, 2003)

Yang GUO, Zihui GE, Bhuvan URGAONKAR, Prashant SHENOY, Don TOWSLEY.

Dynamic Cache Reconfiguration Strategies for a Cluster-Based Streaming Proxy.

Web Content Caching And Distribution: Proceedings Of The 8th International Workshop, IBM T.J. Watson Research Center, Hawthorne, New York, USA, September 29 - October 1st, 2003.

(Hada et al, 1999)

Hisakazu HADA, Ken ichi CHINEN, Suguru YAMAGUCHI, and Yugi OIE. Behaviour of WWW proxy servers in low bandwidth conditions. In Proceedings of the 4th International Web Caching Workshop, April 1999. http://www.ircache.net/Cache/Workshop99/Papers/hada-0.ps.gz

(Harvest)

Harvest project. http://webharvest.sourceforge.net/ng/

(HTTP 1.1)

R. FIELDING, J. GETTYS, J.C. MOGUL, H. FRYSTYK, T. BERNERS-LEE. Hypertext transfer protocol - HTTP/1.1. November 21, 1997. ftp://ietf.org/internet-drafts/draft-ietf-http-v11-spec-rev-01.txt

(ICP)

Internet Cache Protocol. http://icp.ircache.net/

(IRCache)

IRCache project. http://www.ircache.net/

(Jin et al, 2002)

Shudong JIN, Azer BESTAVROS, Arun IYENGAR. Accelerating Internet Streaming Media Delivery using Network-Aware Partial Caching. Proceedings of 22nd International Conference on Distributed Computing Systems, 2002. Page(s): 153- 160

(Johnson)

Ervin JOHNSON. Increasing the Performance of Transparent Caching with Content-aware Cache Bypass. ArrowPoint Communications, Westford, MA.

(Kafri and Janeček, 2002)

Nedal KAFRI, Jan JANEČEK.
Dynamic Behaviour of the Distributed Tree Quorum Algorithm.
Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), 2002.

(Kangasharju et al, 2000)

Jussi KANGASHARJU, James ROBERTS, Keith W. ROSS.
Performance Evaluation of Redirection Schemes in Content Distribution Networks.

Proceedings of 5th International Web Caching and Content Delivery Workshop.

(Kanodia and Knightly, 2003)

Vikram KANODIA and Edward W. KNIGHTLY, Member, IEEE.

Ensuring Latency Targets in Multiclass Web Servers.

IEEE Transactions on Parallel and Distributed Systems. Volume: 14, Issue: 1, Year: January 2003. Page(s): 84-93.

(Karger et al, 1999)

David KARGER, Alex SHERMAN, Andy BERKHEIMER, Bill BOGSTAD, Rizwan DHANIDINA, Ken IWAMOTO, Brian KIM, Luke MATKINS, and Yoav YERUSHALMI.

Web caching with consistent hashing.

In Proceedings of the 8th International WWW Conference, Toronto, Canada, May 1999. http://www8.org/w8-papers/2a-webserver/caching/paper2.html

(Kelly, 2001)

Terence KELLY.

Thin-Client Web Access Patterns: Measurements from a Cache-Busting Proxy.

Proceedings of the Sixth International Workshop on Web Caching and Content Distribution. Boston University, Boston, Massachusetts, USA. June 20-22, 2001.

(Koch et al, 2002)

KOCH, R.R.; MOSER, L.E.; MELLIAR-SMITH, P.M.

The totem redundant ring protocol.

Page(s): 598- 607. Proceedings of 22nd International Conference on Distributed Computing Systems, 2002.

(Koletsou and Voelker, 2001)

Mimika KOLETSOU, Geoffrey M. VOELKER.

The Medusa Proxy: A Tool For Exploring User-Perceived Web Performance.

Proceedings of the Sixth International Workshop on Web Caching and

Content Distribution. Boston University, Boston, Massachusetts, USA. June 20-22, 2001.

(Kopparapu)

Chandra KOPPARAPU.

Load Balancing Servers, Firewalls, and Caches.

John Wiley & Sons.

(Kondou et al, 2002)

Daisuke KONDOU, Hideo MASUDA, Toshimitsu MASUZAWA.

A self-stabilizing protocol for pipelined PIF in tree networks.

Proceedings of 22nd International Conference on Distributed Computing Systems, 2002. Page(s): 181- 190.

(Krishnamurthy and Wills, 2000)

Balachander KRISHNAMURTHY and Craig E. WILLS.

Analyzing factors that influence end-to-end Web performance. In Proceedings of the 9th International WWW Conference, May 2000. http://www9.org/w9cdrom/371/371.html

(Krishnan et al, 1999)

P. KRISHNAN, Danny RAZ, Juval SHAVITT.

Transparent En-Route caching in WANs?

In Proceedings of the 4th International Web Caching Workshop, April 1999.

(Kweon and Shin, 2003)

Seok-Kyu KWEON, SHIN K.G.

Statistical real-time communication over Ethernet.

IEEE Transactions on Parallel and Distributed Systems. Volume: 14, Issue: 4, Year: April 2003. Page(s): 322- 335

(Li et al, 1999)

Bo LI, Mordecai GOLIN, Giuseppe ITALIANO, Xin DENG, and Kazem SOHRABY.

On the optimal placement of Web proxies in the Internet.

In Proceedings of the INFOCOM '99 conference, March 1999.

http://www.ieee-infocom.org/1999/papers/09d_01.pdf

(Lee and Tomlinson, 1999)

Ron LEE, Gary TOMLINSON.

Workload Requirements for a Very High-Capacity Proxy Cache Design.

The 4th International Web Caching Workshop. San Diego, California, March 31 - April 2, 1999.

(Maltzahn et al, 1999)

Carlos MALTZAHN, Kathy RICHARDSON, and Dirk GRUNWALD. Reducing the disk I/O of Web proxy server caches.

In Proceedings if the USENIX Annual Technical Conference, June 1999.

(Markatos et al, 1999)

Evangelos P. MARKATOS, Manolis G.H. KATEVENIS, Dionisis PNEVMATIKATOS, and Michail FLOURIS.

Secondary storage management for Web proxies.

In Proceedings of the 1999 Usenix Symposium on Internet Technologies and Systems (USITS'99), October 1999.

http://archvlsi.ics.forth.gr/papers/1999.USITS99.web_proxy_storage.ps.gz

(Menasce, 2002)

Daniel A. MENASCE. Trade-offs in Designing Web Clusters. IEEE Internet Computing, 6(5), September/October 2002.

http://computer.org/internet/ic2002/w5076abs.htm

(McKusick et al, 1996)

Marshall Kirk MCKUSICK, Keith BOSTIC, Michael J. KARELS, John S. QUARTERMAN. The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley Longman, 1996.

(McKusick et al, 2004)

Marshall Kirk MCKUSICK, George NEVILLE-NEIL. The Design and Implementation of the FreeBSD Operating System. Addison Wesley Professional, 2004.

(mail.ru)

mail.ru – Biggest Russian free email service.

(Merwe et al, 2003)

Jacobus Van der MERWE, Paul GAUSMAN, Chuck CRANOR, Rustam AKHMAROV.

Design, Implementation and Operation of a large Enterprise Content Distribution Network.

Web Content Caching And Distribution: Proceedings Of The 8th International Workshop, IBM T.J. Watson Research Center, Hawthorne, New York, USA, September 29 - October 1, 2003.

(NetApp)

Network Appliance, Inc. (NetApp). http://www.netapp.com/

(Netscape autoconfig)

Netscape. Proxy auto-configuration.

http://search.netscape.com/eng/mozilla/2.0/relnotes/demo/proxy-live.htm

(NLANR)

National Laboratory for Applied Network Research.

http://www.nlanr.net

(Orman, 2001)

Hilarie K. ORMAN.

Data Integrity for Mildly Active Content. Proceedings of Third International Workshop on Active Middleware Services. San Francisco, California. August 6th 2001.

(Panteleenko and Freeh, 2001)

Vsevolod V. PANTELEENKO, Vincent W. FREEH.

Instantaneous Offloading of Transient Web Server Load.

Proceedings of the Sixth International Workshop on Web Caching and Content Distribution. Boston University, Boston, Massachusetts, USA. June 20-22, 2001.

(Polygraph)

Web Polygraph. Performance benchmark for web-caches. http://polygraph.ircache.net/

(Rabinovich et al, 2003)

Michael RABINOVICH, Zhen XIAO, Amit AGGARWAL.

Computing on the Edge: A Platform for Replicating Internet Applications. Web Content Caching And Distribution: Proceedings Of The 8th International Workshop, IBM T.J. Watson Research Center, Hawthorne, New York, USA, September 29 - October 1st, 2003.

(Rabinovich and Spatschak)

Michael RABINOVICH and Oliver SPATSCHAK.

Web Caching and Replication.

Addison Wesley Professional.

(Ramaswamy and Liu, 2002)

Lakshmish RAMASWAMY, Ling LIU.

A new document placement scheme for cooperative caching on the Internet. Proceedings of 22nd International Conference on Distributed Computing Systems, 2002. Page(s): 95- 103

(RFC2186)

Kim CLAFFY, Duane WESSELS.

RFC 2186. Internet Cache Protocol (ICP), version 2. http://ds.internic.net/rfc/rfc2186.txt

(Riska et al, 2002)

Alma RISKA, Wei SUN, Evgenia SMIRNI, Gianfranco CIARDO. AdaptLoad: effective balancing in clustered web servers under transient load conditions.

Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), 2002.

(Rizzo and Vicisano, 1998)

Luigi RIZZO and Lorenzo VICISANO.

Replacement policies for a proxy cache.

Technical Report RN/98/13, UCL-CS, 1998. http://www.iet.unipi.it/~luigi/lrv98.ps.gz

(Rodriguez et al, 1999)

Pablo RODRIGUEZ, Christian SPANNER, Ernst W. BIERSACK.

Web Caching Architectures: Hierarchical and Distributed Caching.

The 4th International Web Caching Workshop. San Diego, California, March 31 - April 2, 1999.

(Rost et al, 2001)

Stanislav ROST, John BYERS, Azer BESTAVROS.

The Cyclone Server Architecture: Streamlining Delivery of Popular Content.

Proceedings of the Sixth International Workshop on Web Caching and Content Distribution.

Boston University, Boston, Massachusetts, USA. June 20-22, 2001.

(Rousskov and Wessels, 1998)

Alex ROSSSKOV, Duane WESSELS.

Cache Digests.

Presented at the 3rd International WWW Caching Workshop, Manchester, England, June 1998.

http://wwwcache.ja.net/events/workshop/31/rousskov@nlanr.net.ps

(RUNNet)

Russian Federal University Computer Network RUNNet. http://www.runnet.ru/

(Schmid et al, 2002)

Ulrich SCHMID, Bettina WEISS, John RUSHBY.

Formally Verified Byzantine Agreement in Presence of Link Faults.

Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), 2002.

(Schroeder et al, 2000)

Trevor SCHROEDER, Steve GODDARD, and Byrav RAMAMURTHY.

Scalable Web Server Clustering Technologies.

IEEE Network, May/June 2000.

http://www.comsoc.org/ni/private/2000/may/Goddard.html

(Selvakumar and Prabhakar, 2001)

S. SELVAKUMAR and P. PRABHAKAR.

Implementation and comparison of distributed caching schemes.

Computer Communications, 24(7-8):677-684, April 2001

(Squid)

Squid Web Proxy Cache.

http://www.squid-cache.org/

(Stevens)

W. Richard STEVENS.

UNIX Network Programming.

Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

(Spreitzer and Janssen, 2000)

Mike SPREITZER and Bill JANSSEN.

HTTP 'next generation'.

In Proceedings of the 9th International WWW Conference, May 2000.

(Sultan et al., 2002)

Florin SULTAN, Kiran SRINIVASAN, Deepa IYER, Liviu IFTODE. Migratory TCP: connection migration for service continuity in the Internet. Proceedings of 22nd International Conference on Distributed Computing Systems, 2002. Page(s): 469- 470.

(Vaidya and Christensen, 2001)

Sujit VAIDYA and Kenneth J. CHRISTENSEN.

A Single System Image Server Cluster using Duplicated MAC and IP Addresses.

In Proceedings of the IEEE Conference on Local Computer Networks, Tampa, FL USA, November 2001.

(W3C)

WWW Consortium. Propagation, Caching and Replication on the Web.

http://www.w3.org/Propagation/

(Wessels)

Web Caching. Duane WESSELS.

O'Reilly and Associates.

(Williams, 1998)

Bert WILLIAMS.

Transparent Web Caching Solutions.

3rd International WWW Caching Workshop. University of Manchester. Manchester, England, June 15-17 1998.

(Wisconsin)

Wisconsin Proxy Benchmark. http://www.cs.wisc.edu/~cao/wpb1.0.html

(Yuan et al, 2003)

Chun YUAN, Zhigang HUA, Zheng ZHANG.

Proxy+: Simple Proxy Augmentation for Dynamic Content Processing.

Web Content Caching And Distribution: Proceedings Of The 8th International Workshop, IBM T.J. Watson Research Centre, Hawthorne, New York, USA, September 29 - October 1st, 2003.

(Zipf, 1949)

G. ZIPF.

Human Behaviour and the Principle of Least Effort.

Addison Wesley, 1949.

APPENDIX 1. PACKET FILTER

Packet Filter is implemented as part of FreeBSD ipfw module. It is quite simple (partly because it uses existing parts of ipfw module) but not very efficient. However for research purposes it is good enough. Building high-performance packet filter would become one of the important tasks during implementation of production-level software. Here are modifications to source code of original FreeBSD ipfw module presented in patch format:

```
*** ip_fw2.c.orig Tue Oct 26 09:54:43 2004
--- ip_fw2.c      Tue Oct 26 09:51:26 2004
***************
*** 149,154 ****
--- 149,166 ----
  SYSCTL_INT(_net_inet_ip_fw, OID_AUTO, verbose_limit, CTLFLAG_RW,
      &verbose_limit, 0, "Set upper limit of matches of ipfw rules
logged");

+ /* Cluster sysctl variables start here */
+
+ static unsigned cl_extifip;
+ static unsigned cl_bitmask[16];
+
+ SYSCTL_INT(_net_inet_ip_fw, OID_AUTO, cl_extifip, CTLFLAG_RW,
+     &cl_extifip, 0, "Cluster external interface address");
+ SYSCTL_OPAQUE(_net_inet_ip_fw, OID_AUTO, cl_bitmask, CTLFLAG_RW,
+     &cl_bitmask, sizeof(cl_bitmask), 0, "Cluster node bitmask");
+
+ /* Cluster sysctl variables end here */
+
  /*
   * Description of dynamic rules.
   *
***************
*** 423,428 ****
--- 435,474 ----
      return(0);  /* no match, fail ... */
  }
```

```
+ /* Cluster support routines */
+ /*
+ * 2 sysctl variables are used:
+ *   cl_extifip  integer          IP address of ext. interface
+ *   cl_bitmask  opaque           Node bitmask
+ */
+
+ static int
+ iface_match_by_ip(struct ifnet *ifp, unsigned ip_a) /* XXX */
+ {
+     struct ifaddr *ia;
+
+     TAILQ_FOREACH(ia, &ifp->if_addrhead, ifa_link) {
+           if (ia->ifa_addr == NULL)
+                 continue;
+           if (ia->ifa_addr->sa_family != AF_INET)
+                 continue;
+           if (ip_a == ((struct sockaddr_in *)
+               (ia->ifa_addr))->sin_addr.s_addr)
+                 return(1);  /* match */
+     }
+
+     return(0);
+ }
+
+ static int
+ cl_pf_hash(unsigned ip_a) /* XXX */
+ {
+     return (((ip_a & 0xff) + ((ip_a & 0xff00)>>8) +
+           ((ip_a & 0xff0000)>>16) + ((ip_a & 0xff000000)>>24)) % 512);
+ }
+
+ /* End of cluster support routines */
+
  /*
   * The 'verrevpath' option checks that the interface that an IP packet
   * arrives on is the same interface that traffic destined for the
```

```
***************
*** 1656,1661 ****
--- 1702,1735 ----
      args->f_id.src_port = src_port = ntohs(src_port);
      args->f_id.dst_port = dst_port = ntohs(dst_port);

+     /*
+     * Web-cache cluster packet filter starts here
+     * Uses 2 variables:
+     *    cl_extifip          IP address of incoming interface
+     *    cl_bitmask          Packet filter bitmask
+     */
+
+     /* Only TCP is relevant for packet filter */
+     if (proto != IPPROTO_TCP)
+          goto after_ip_checks;
+     /* Assure that packet is incoming */
+     if (iface_match_by_ip(m->m_pkthdr.rcvif, cl_extifip) != 1)
+          goto after_ip_checks;
+     /* Check that packet dest. IP is equal to ext. interface IP */
+     if (args->f_id.dst_ip == cl_extifip) {
+          if (args->f_id.dst_ip == args->f_id.src_ip)
+               return(IP_FW_PORT_DENY_FLAG);
+          hash = cl_pf_hash(args->f_id.src_ip);
+     } else {
+          hash = cl_pf_hash(args->f_id.dst_ip);
+     }
+
+     if ((cl_bitmask[hash/32] & (1 >> (hash%32))) == 0)
+          return(IP_FW_PORT_DENY_FLAG);
+
+     /*
+     * Web-cache cluster packet filter ends here
+     */
+
 after_ip_checks:
     if (args->rule) {
          /*
```

User-mode interface library is also quite simple, its main purpose is to hide machine-dependant details of implementation, so that during porting to other Operating System only internal part of the library and packet filter itself would need to be changed. Here is source code of library – both interface and implementation.

```
/* $Id: PacketFilter.h$
* Packet Filter interface. Hides machine-dependant details of
implementation.
*/

#ifndef __PACKET_FILTER_H_
#define __PACKET_FILTER_H_

#include <string>

#include "Config.h"
#include "Bitmask.h"
#include "LogFile.h"

class PacketFilter {

public:
      PacketFilter(unsigned ip_a = 0) { reset(); machdep_set_ip(ip_a); }
      ~PacketFilter() { reset(); }
      const Bitmask &get()
      {
            unsigned ary[CL_BITMASK_INTS];
            machdep_get_mask(ary);
            bitmask = Bitmask(ary);
            return bitmask;
      }
      void set_mask(Bitmask &bm)
      {
            bitmask = bm;
            machdep_set_mask(bitmask.get_unsigned_ptr());
      }
```

```
        void reset()
        {
                bitmask.zero();
                machdep_set_mask(bitmask.get_unsigned_ptr());
        }

private:
        void machdep_set_mask(const unsigned *bm);
        void machdep_set_ip(unsigned ip_a);
        void machdep_get_mask(unsigned *bm);

        Bitmask bitmask;
};

#endif /* __PACKET_FILTER_H_ */

/* $Id: packet_filter.cpp$
* User-level interface to packet filter. Hides machine-dependant details
* of implementation, and is included into cluster software build process.
*/

#include <sys/types.h>
#include <sys/sysctl.h>

#include "Config.h"
#include "LogFile.h"
#include "PacketFilter.h"

void
PacketFilter::machdep_set_mask(const unsigned *bm)
{
        ASSERT(bm != NULL);

#ifdef __FreeBSD__
        int mib[5], ret;
        size_t len;
```

```
        /* Fill out the first three components of the mib */
        len = 5;
        ret = sysctlnametomib("net.inet.ip.fw.cl_bitmask", mib, &len);
        if (ret)
                die("sysctlnametomib");

        /* Set new variable */
        if (sysctl(mib, 5, NULL, NULL, (void *)bm, CL_BITMASK_SIZEOF) == -1)
                die("sysctl");
#endif /* __FreeBSD__ */
}

void
PacketFilter::machdep_set_ip(unsigned ip_a)
{
#ifdef __FreeBSD__
        int mib[5], ret;
        size_t len;

        /* Fill out the first three components of the mib */
        len = 5;
        ret = sysctlnametomib("net.inet.ip.fw.cl_extifip", mib, &len);
        if (ret)
                die("sysctlnametomib");

        /* Set new variable */
        if (sysctl(mib, 5, NULL, NULL, (void *)&ip_a, 4) == -1)
                die("sysctl");
#endif
}

void
PacketFilter::machdep_get_mask(unsigned *bm)
{
        ASSERT(bm != NULL);

#ifdef __FreeBSD__
        int mib[5], ret;
```

```
        size_t len;


        /* Fill out the first three components of the mib */
        len = 5;
        ret = sysctlnametomib("net.inet.ip.fw.cl_bitmask", mib, &len);
        if (ret)
                die("sysctlnametomib");

        /* Set new variable */
        len = CL_BITMASK_SIZEOF;
        if (sysctl(mib, 5, (void *)bm, &len, NULL, 0) == -1)
                die("sysctl");
#endif /* __FreeBSD__ */
}
```

APPENDIX 2. BITMASK MANIPULATION

```
/* $Id: Bitmask.h$
* Bitmask operations interface.
*/

#ifndef __BITMASK_H_
#define __BITMASK_H_

#include "Config.h"

#include <string.h>
#include <string>

class Bitmask {

private:
      unsigned ary[CL_BITMASK_INTS];

public:
      void zero() { bzero((void *)ary, CL_BITMASK_SIZEOF); }
      void one() { memset((void *)ary, 0xff, CL_BITMASK_SIZEOF); }

      Bitmask() { zero(); }
      Bitmask(const Bitmask & bm) { memcpy((void *)ary, bm.ary,
CL_BITMASK_SIZEOF); }
      Bitmask(const unsigned *bm);

      bool is_zero();
      bool is_one();

      const unsigned *get_unsigned_ptr() { return ary; }
      std::string get_string();

      bool operator== (const Bitmask &b);
```

```
        bool operator!= (const Bitmask &b);
        Bitmask operator& (const Bitmask &b);
        Bitmask operator| (const Bitmask &b);
        Bitmask operator- (const Bitmask &b);
        Bitmask operator|= (const Bitmask &b);
        Bitmask operator-= (const Bitmask &b);
        Bitmask operator~();

        Bitmask get_one_bit();
};


#endif /* __BITMASK_H_ */


/* $Id: bitmask.cpp$
* Bitmask class implementation
*/

#include <string.h>

#include "Config.h"
#include "Bitmask.h"
#include "Util.h"

bool
Bitmask::is_zero()
{
        for (int i = 0; i < CL_BITMASK_INTS; i++) {
                if ((ary[i] & 0xffffffff) != 0)
                        return false;
        }

        return true;
}

bool
Bitmask::is_one()
```

```
{
      for (int i = 0; i < CL_BITMASK_INTS; i++) {
            if ((ary[i] & 0xffffffff) != 0xffffffff)
                  return false;
      }

      return true;
}

Bitmask::Bitmask(const unsigned *bm)
{
      if (bm == NULL) {
            zero();
            return;
      }
      memcpy((void *)ary, bm, CL_BITMASK_SIZEOF);
}

std::string
Bitmask::get_string()
{
      std::string s;

      for (int i = 0; i < CL_BITMASK_INTS; i++) {
            s += Sprintf("%x", ary[i]);
      }

      return s;
}

bool
Bitmask::operator== (const Bitmask &b)
{
      for (int i = 0; i < CL_BITMASK_INTS; i++)
            if (ary[i] != b.ary[i])
                  return false;

      return true;
```

```
}

bool
Bitmask::operator!= (const Bitmask &b)
{
      for (int i = 0; i < CL_BITMASK_INTS; i++)
            if (ary[i] != b.ary[i])
                  return true;

      return false;
}

Bitmask
Bitmask::operator& (const Bitmask &b)
{
      unsigned res[CL_BITMASK_INTS];

      for (int i = 0; i < CL_BITMASK_INTS; i++)
            res[i] = ary[i] & b.ary[i];

      return Bitmask(res);
}

Bitmask
Bitmask::operator| (const Bitmask &b)
{
      unsigned res[CL_BITMASK_INTS];

      for (int i = 0; i < CL_BITMASK_INTS; i++)
            res[i] = ary[i] | b.ary[i];

      return Bitmask(res);
}

Bitmask
Bitmask::operator- (const Bitmask &b)
{
      unsigned res[CL_BITMASK_INTS];
```

```
        for (int i = 0; i < CL_BITMASK_INTS; i++)
                res[i] = ary[i] & ~b.ary[i];

        return Bitmask(res);
}

Bitmask
Bitmask::operator|= (const Bitmask &b)
{
        for (int i = 0; i < CL_BITMASK_INTS; i++)
                ary[i] = ary[i] | b.ary[i];

        return *this;
}

Bitmask
Bitmask::operator-= (const Bitmask &b)
{
        for (int i = 0; i < CL_BITMASK_INTS; i++)
                ary[i] = ary[i] & ~b.ary[i];

        return *this;
}

Bitmask
Bitmask::operator~ ()
{
        unsigned res[CL_BITMASK_INTS];

        for (int i = 0; i < CL_BITMASK_INTS; i++)
                res[i] = ~ary[i];

        return Bitmask(res);
}

Bitmask
Bitmask::get_one_bit()
```

```
{
        Bitmask ret;
        static const unsigned masks[] = {
                0x80000000, 0x40000000, 0x20000000, 0x10000000,
                0x8000000, 0x4000000, 0x2000000, 0x1000000,
                0x800000, 0x400000, 0x200000, 0x100000,
                0x80000, 0x40000, 0x20000, 0x10000,
                0x8000, 0x4000, 0x2000, 0x1000,
                0x800, 0x400, 0x200, 0x100,
                0x80, 0x40, 0x20, 0x10,
                0x8, 0x4, 0x2, 0x1
        };

        for (int i = 0; i < CL_BITMASK_INTS; i++) {
                unsigned val = ary[i];
                if (!val)
                        continue;
                for (int j = 0; j < 32; j++) {
                        if (val & masks[j]) {
                                ret.ary[i] = masks[j];
                                return ret;
                        }
                }
        }

        return ret;
}



/* $Id: Cluster.h$
* Cluster-state structures
*/

#ifndef __CLUSTER_H_
#define __CLUSTER_H_

#include <sys/time.h>
```

```
#include "Config.h"
#include "Bitmask.h"
#include "CSP.h"


/*
* Node's states
*/
#define NODE_ACTIVE          1
#define NODE_PASSIVE         2
#define NODE_OFFLINE         3
#define NODE_FAILED          4
#define NODE_DEAD            5

/*
* Node information structure
*/
class NodeState {

public:
      unsigned    id;
      int         state;
      double            update_time;
      unsigned    load;
      Bitmask           bitmask;

      NodeState() : id(0), state(NODE_OFFLINE), update_time(0.0), load(0),
bitmask() {}
};

class ClusterState {

public:
      unsigned    my_id;
      unsigned    my_ext_ip;
      int         my_state;
      double            last_load_redist;
```

```
        NodeState   nodes[CL_MAX_NODES];

        ClusterState() : my_id(0), my_ext_ip(0), my_state(NODE_OFFLINE),
last_load_redist(0.0) {}
};


struct BitmaskCommand {
        BitmaskCommand() : node_id(0), bitmask() {}
        BitmaskCommand(unsigned id, const Bitmask &bm) : node_id(id),
bitmask(bm) {}

        unsigned node_id;
        Bitmask bitmask;
};

namespace glob {
        extern ClusterState state;
};


#endif /* __CLUSTER_H_ */


/* $Id: bitmasks.cpp$
* Bitmask recovery and load redistribution procedures
*/

#include <limits.h>
#include <vector>

#include "Bitmask.h"
#include "LogFile.h"
#include "Cluster.h"


namespace glob {
```

```
        ClusterState state;
};

std::vector<BitmaskCommand>
get_recovery_vector()
{
        /* Copy global state */

        Bitmask corr_bm;
        ClusterState new_st = glob::state;

        /* Eliminate instersections in pairs */

        for (int i = 0; i < CL_MAX_NODES-1; i++) {
                for (int j = i+1; j < CL_MAX_NODES; j++) {
                        Bitmask bm = new_st.nodes[i].bitmask &
new_st.nodes[j].bitmask;
                        if (!bm.is_zero()) {
                                new_st.nodes[i].bitmask -= bm;
                                new_st.nodes[j].bitmask -= bm;
                                corr_bm |= bm;
                        }

                }
        }

        /* Eliminate empty space in combined bitmask */
        Bitmask sum;
        for (int i = 0; i < CL_MAX_NODES; i++) {
                if (new_st.nodes[i].state == NODE_ACTIVE ||
new_st.nodes[i].state == NODE_PASSIVE)
                        sum |= new_st.nodes[i].bitmask;
        }
        corr_bm |= ~sum;

        /* Move load from nodes in FAILED or OFFLINE state*/
        for (int i = 0; i < CL_MAX_NODES; i++) {
```

```
            if (new_st.nodes[i].state != NODE_ACTIVE &&
new_st.nodes[i].state != NODE_PASSIVE) {
                  corr_bm |= new_st.nodes[i].bitmask;
                  new_st.nodes[i].bitmask.zero();
            }
      }

      /* Apply load to least loaded node */
      /* XXX it would be more efficient to split load across several nodes
*/
      unsigned min_load = UINT_MAX;
      int min_loaded_ix = -1;
      for (int i = 0; i < CL_MAX_NODES; i++) {
            if (new_st.nodes[i].state == NODE_ACTIVE ||
new_st.nodes[i].state == NODE_PASSIVE) {
                  if (new_st.nodes[i].load <= min_load) {
                        min_load = new_st.nodes[i].load;
                        min_loaded_ix = i;
                  }
            }
      }
      ASSERT(min_loaded_ix >= 0);
      new_st.nodes[min_loaded_ix].bitmask |= corr_bm;

      /* Create command vector */
      std::vector<BitmaskCommand> vec;
      for (int i = 0; i < CL_MAX_NODES; i++) {
            if (new_st.nodes[i].state == NODE_ACTIVE ||
new_st.nodes[i].state == NODE_PASSIVE) {
                  if (glob::state.nodes[i].bitmask !=
new_st.nodes[i].bitmask) {

      vec.push_back(BitmaskCommand(glob::state.nodes[i].id,
new_st.nodes[i].bitmask));
                  }
            }
      }
```

```
        return vec;
}

std::vector<BitmaskCommand>
get_load_redist_vector()
{
        std::vector<BitmaskCommand> ret;

        /* Find least loaded and most loaded nodes */

        unsigned min_load = UINT_MAX, max_load = 0;
        int min_loaded_ix = -1, max_loaded_ix = -1;
        for (int i = 0; i < CL_MAX_NODES; i++) {
                if (glob::state.nodes[i].state == NODE_ACTIVE ||
glob::state.nodes[i].state == NODE_PASSIVE) {
                        if (glob::state.nodes[i].load <= min_load) {
                                min_load = glob::state.nodes[i].load;
                                min_loaded_ix = i;
                        }
                        if (glob::state.nodes[i].load >= max_load) {
                                max_load = glob::state.nodes[i].load;
                                max_loaded_ix = i;
                        }
                }
        }
        ASSERT(min_loaded_ix >= 0 && max_loaded_ix >= 0);

        /* Do nothing if difference is not so big */
        if (max_load < 2*min_load)
                return ret;

        /* Move one bit of load from least loaded to most loaded node */
        Bitmask one_bit =
glob::state.nodes[max_loaded_ix].bitmask.get_one_bit();
        Bitmask new_least_loaded =
glob::state.nodes[min_loaded_ix].bitmask.get_one_bit() | one_bit;
        Bitmask new_most_loaded =
glob::state.nodes[max_loaded_ix].bitmask.get_one_bit() - one_bit;
```

```
        /* Create vector */
        ret.push_back(BitmaskCommand(glob::state.nodes[max_loaded_ix].id,
new_most_loaded));
        ret.push_back(BitmaskCommand(glob::state.nodes[min_loaded_ix].id,
new_least_loaded));

        return ret;
}
```